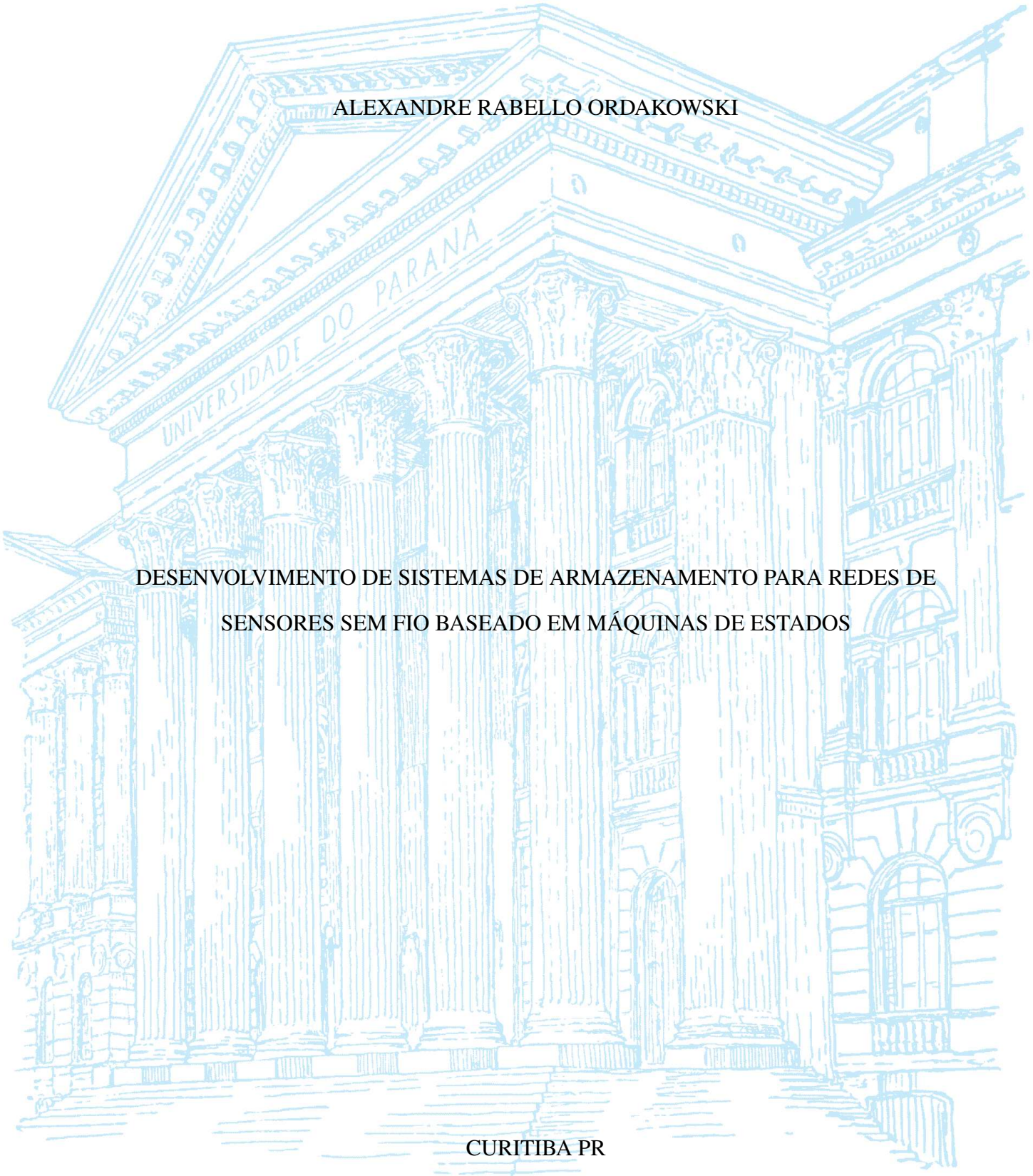
UNIVERSIDADE FEDERAL DO PARANÁ

ALEXANDRE RABELLO ORDAKOWSKI



DESENVOLVIMENTO DE SISTEMAS DE ARMAZENAMENTO PARA REDES DE SENSORES SEM FIO BASEADO EM MÁQUINAS DE ESTADOS

CURITIBA PR

2021

ALEXANDRE RABELLO ORDAKOWSKI

DESENVOLVIMENTO DE SISTEMAS DE ARMAZENAMENTO PARA REDES DE SENSORES SEM FIO BASEADO EM MÁQUINAS DE ESTADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Carmem Satie Hara.

Coorientador: Marcos Aurélio Carrero.

CURITIBA PR

2021

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

O65d Ordakowski, Alexandre Rabello
Desenvolvimento de sistemas de armazenamento para redes de sensores sem fio baseado em máquinas de estados [recurso eletrônico] / Alexandre Rabello Ordakowski. – Curitiba, 2021.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2021.

Orientador: Carmem Satie Hara – Coorientador: Marcos Aurélio Carrero

1. Armazenamento de dados. 2. Sistemas de comunicação sem fio. 3. Software - Desenvolvimento. I. Universidade Federal do Paraná. II. Hara, Carmem Satie. III. Carrero, Marcos Aurélio. IV. Título.

CDD: 004.568

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **ALEXANDRE RABELLO ORDAKOWSKI** intitulada: **DESENVOLVIMENTO DE SISTEMAS DE ARMAZENAMENTO PARA REDES DE SENSORES SEM FIO BASEADO EM MÁQUINAS DE ESTADOS**, sob orientação da Profa. Dra. CARMEM SATIE HARA, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 17 de Dezembro de 2020.

Assinatura Eletrônica
18/12/2020 08:45:15.0
CARMEM SATIE HARA
Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
21/12/2020 09:59:13.0
CRISTINA DUTRA DE AGUIAR CIFERRI
Avaliador Externo (INSTITUTO DE CIêNCIAS MATEMáTICAS E DE COMPUTAçãO ICMC - USP)

Assinatura Eletrônica
18/12/2020 06:44:13.0
ALDRI LUIZ DOS SANTOS
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Não há exemplo maior de dedicação do que o da nossa família. À minha querida família, que tanto admiro, dedico o resultado do esforço realizado ao longo deste percurso.

AGRADECIMENTOS

Agradeço aos meus pais Laudair e Margareth, pelo apoio e incentivo nas horas difíceis. A minha esposa Luana, por todo amor e apoio, e também por compreender minha ausência durante o tempo dedicado aos estudos. Também agradeço aos meus irmãos, avós, tios, primos e amigos que de alguma forma também contribuíram para realização deste sonho. Aos professores, pelos ensinamentos e correções que me permitiram apresentar um melhor desempenho na minha formação profissional. As professores orientadores Carmem e Marcos, pelo seu grande desprendimento em ajudar e amizade sincera. Por fim, agradeço a todos que contribuíram diretamente ou indiretamente durante a formação.

RESUMO

A crescente demanda por dispositivos sensores, elementos-chave dos sistemas ciber-físicos (CPS) e da Internet das Coisas (IoT), exige que novos modelos de armazenamento sejam desenvolvidos para lidar com o grande volume de dados gerados. Porém, a especificação e implementação de tais sistemas é uma tarefa complexa, principalmente pela falta de suporte para reutilização de código e pela dificuldade em definir o fluxo de execução. Para resolver este problema, nesta dissertação é apresentado o SMDM-SD (*State Machine Development Model for Sensor Devices*), um modelo de desenvolvimento de software para redes de sensores sem fio (RSSF) baseado em máquinas de estados. No SMDM-SD o programador projeta o fluxo geral de execução do sistema como uma máquina de estados com transições lógicas e transições baseadas em eventos. Para a implementação da máquina, foi desenvolvida uma linguagem específica de domínio, chamada SLEDS-SD (*State Machine-based Language for Event-Driven Systems for Sensor Devices*), com estruturas de controle que se assemelham às transições usadas na fase de projeto. Um programa SLEDS-SD pode ser visto como uma orquestração de serviços fornecidos por componentes reutilizáveis, associados às entidades da aplicação. Em sua implementação atual, o SLEDS-SD gera código nesC, que pode ser instalado em dispositivos baseados em TinyOS. A avaliação do SMDM-SD envolveu o desenvolvimento de três modelos de armazenamento. A eficiência da proposta foi avaliada determinando a quantidade de reutilização de código promovida pelo modelo. Além disso, sua eficácia foi avaliada comparando as funcionalidades dos sistemas resultantes com as relatadas em outros estudos.

Palavras-chave: Modelo de Desenvolvimento. Sistemas de Armazenamento. Rede de Sensores.

ABSTRACT

The growing demand for sensor devices, key elements of cyber-physical systems (CPS) and the Internet of Things (IoT), requires that new models of storage systems be proposed to deal with the huge volume of generated data. However, the specification and implementation of such systems is a complicated task, especially for the lack of support for code reuse and the difficulty in defining the execution flow. To address this problem, in this dissertation we present SMDM-SD (*State Machine Development Model for Sensor Devices*), a software development model for wireless sensor networks (WSN) based on state machines, that supports the development of storage systems. In SMDM-SD the programmer designs the overall execution flow of the system as a state machine with both logical and event-based transitions. For the implementation of the state machine, we have developed a domain specific language, called SLEDS-SD (*State Machine-based Language for Event-Driven Systems for Sensor Devices*), with control structures that closely resembles the transitions used in the design phase. A SLEDS-SD program can be seen as an orchestration of services provided by reusable components associated with application entities. In its current implementation, SLEDS-SD generates nesC code, which can be installed in TinyOS-based devices. The evaluation involved the development of three storage models. The efficiency of the proposal was evaluated by deterining the amount of code reuse promoted by the model. Its efficacy was evaluated by comparing the resulting systems performance with those reported in previous studies.

Keywords: Development Model. Storage Systems. Sensor Network.

LISTA DE FIGURAS

2.1 Topologias arquitetônicas na perspectiva das RSSFs. Adaptado de (Rashid e Rehmani, 2016).. 15
2.2 Componentes de um nó sensor. Adaptado de (Ruiz, 2003). 16
2.3 Classificação dos modelos de armazenamento em RSSF. 18
2.4 Roteamento tradicional e roteamento centrado em dados. Adaptado de (Loureiro et al., 2003).. 19
2.5 Máquina de estados para representação do fluxo de execução de um sistema. . . . 20
2.6 Desenvolvimento de software baseado em componentes. (Lau e di Cola, 2017) . 20
2.7 Principais Elementos do MDD. (Lucrédio, 2009) 21
2.8 Exemplo de árvore sintática. Adaptado de (Aho et al., 1998). 24

3.1 Etapas no desenvolvimento de uma aplicação utilizando SenNet. (Salman e Al-Yasiri, 2016) 26
3.2 Fases de desenvolvimento utilizando o Tokenit. (Taherkordi et al., 2015) 27
3.3 Ambiente de desenvolvimento utilizando o *X-Machine*. (Braga, 2012) 28
3.4 Arquitetura do Wiselib. (Baumgartner et al., 2010) 29
3.5 Fases de desenvolvimento utilizando o IoTSuite. (Soukaras et al., 2015) 30
3.6 Modelagem de uma aplicação de detecção de movimentos utilizando o BIP. (Lekidis et al., 2018) 31
3.7 Arquitetura do RCBM. (Carrero et al., 2017). 32

4.1 Modelo de Desenvolvimento SMDM-SD. 35
4.2 Um modelo de máquina de estados para descoberta de vizinhos por inundação. (Carrero et al., 2017). 36

5.1 Máquina de Estados para as etapas *FloodMax* e *FloodMin* da heurística Max-Min. 42

6.1 Sistema para geração de topologias do TOSSIM, NS-2 e NS-3. 48
6.2 Impacto da densidade da rede no número de $Cluster - Heads$. 49
6.3 Impacto da densidade da rede no número de *Cluster-Heads* na aplicação LCA.. . 49
6.4 Impacto da densidade da rede no número de *Cluster-Heads* na aplicação LEACH. 50
6.5 Cluster formado na execução de Max-Min (Amis et al., 2000). 51
6.6 Impacto da densidade da rede no número de *Cluster-Heads* no aplicação Max-Min. 51
6.7 Impacto da densidade da rede com perda de pacotes no número de *Cluster-Heads* na aplicação Max-Min. 52

LISTA DE TABELAS

3.1 Descrição dos trabalhos relacionados. 33

6.1 Principais características dos Modelos de Armazenamento 45
6.2 Reutilização de código ao implementar modelos de armazenamento 46
6.3 Número de estados e linhas de código do coordenador por modelo de armazenamento 46
6.4 Parâmetros da Simulação 48

A.1 Sintaxe do SLEDS-SD 60
C.1 Tabela com entradas do sistemas de geração de RSSI. 65
C.2 Tabela com a saída do sistema de geração de RSSI. 66

LISTA DE ACRÔNIMOS

AST	*Abstract Syntax Tree* - Árvore de Sintaxe Abstrata
CH	*Cluster-Head* - Cabeça/Líder do Cluster
CM	*Cluster Member* - Membro do Cluster
CXM	*Communicating X-Machines*
DSL	*Domain Specific Language* - Linguagem de Domínio Especifico
EB	Estação-Base
ID	Identificado Único
IDE	*Integrated Development Environment* - Ambiente de Desenvolvimento Integrado
IoT	*Internet of Things* - Internet das Coisas
LCA	*Linked Cluster Algorithm* - Algoritmo de Cluster Vinculado
LEACH	*Low-Energy Adaptive Clustering Hierarchy* - Hierarquia de Cluster Adaptável de Baixa Energia
LOC	*Lines of (source) Code* - Linhas de Código
LP	Linguagem de Programação
MDD	*Model-Driven Development* - Desenvolvimento Orientado a Modelos
ME	Máquina de Estados
nesC	*network embedded systems C*
NS-2	*Network Simulator 2*
RCBM	*Reusable Component-based Model* - Modelo Baseado em Componentes Reutilizáveis
RCBM-S	*Reusable Component-based Model for Sensor devices* - Modelo Baseado em Componentes reutilizáveis para Dispositivos Sensores
RSSF	Redes de Sensores Sem Fio
RF	*Radio Frequency* - Frequência de Rádio
SLEDS	*State Machine-based Language for Event-Driven Systems* - Linguagem Baseada em Máquina de Estados para Sistemas Controlados por Eventos
SMDM-SD	*State Machine Development Model for Sensor Devices* - Modelo de Desenvolvimento de Máquina de Estado para Dispositivos Sensores
SO	Sistema Operacional
XMDL	*X-Machine Definition Language*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO 12
1.1 OBJETIVOS E CONTRIBUIÇÕES 13
1.2 ESTRUTURA DA DISSERTAÇÃO 13
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 REDES DE SENSORES SEM FIO 15
2.1.1 Hardware 16
2.1.2 Sistemas Operacionais 17
2.2 MODELOS DE ARMAZENAMENTO EM RSSF 17
2.3 ESPECIFICAÇÃO DE APLICAÇÕES PARA SENSORES 18
2.3.1 Máquina de Estados 19
2.3.2 Componentes 20
2.3.3 Metamodelos 21
2.4 LINGUAGENS DE DESENVOLVIMENTO PARA SENSORES 21
2.4.1 Linguagem de Domínio Específico 23
2.5 CONSIDERAÇÕES FINAIS 24
3 TRABALHOS RELACIONADOS 26
3.1 SENNET 26
3.2 TOKENIT 27
3.3 X-MACHINE 27
3.4 WISELIB 28
3.4.1 CBCWSN 29
3.5 IOTSUITE 29
3.6 MODELO BIP 30
3.7 RCBM 31
3.8 CONSIDERAÇÕES FINAIS 32
4 O MODELO DE DESENVOLVIMENTO SMDM-SD 35
4.1 ETAPAS DE DESENVOLVIMENTO 35
4.2 ETAPA DE ESPECIFICAÇÃO 36
4.3 ETAPA DE PROGRAMAÇÃO 37
4.4 ETAPA DE COMPILAÇÃO E IMPLANTAÇÃO DA APLICAÇÃO NOS SENSORES 38
4.5 CONSIDERAÇÕES FINAIS 38

5 A LINGUAGEM SLEDS-SD 40
5.1 COMPONENTES DA LINGUAGEM 40
5.2 TRADUÇÃO PARA NESC. 41
5.3 CONSIDERAÇÕES FINAIS 44
6 ESTUDO EXPERIMENTAL 45
6.1 IMPLEMENTAÇÃO DO SISTEMA E MODELOS DE ARMAZENAMENTO . 45
6.2 ANÁLISE DE REUSABILIDADE 46
6.3 AVALIAÇÃO DE COMPORTAMENTO. 47
6.3.1 **Avaliação do número médio de *Cluster-Heads* de acordo com a densidade da rede** 48
6.3.2 **Avaliação do modelo LCA** 49
6.3.3 **Avaliação do modelo LEACH** 50
6.3.4 **Avaliação do modelo MAX-MIN** 50
6.4 CONSIDERAÇÕES FINAIS 52
7 CONCLUSÃO 54
7.1 LISTA DE PUBLICAÇÕES RELACIONADAS À DISSERTAÇÃO 54
7.2 TRABALHOS FUTUROS 55
REFERÊNCIAS 56
APÊNDICE A – SINTAXE DA LINGUAGEM SLEDS SD. 60
APÊNDICE B – CÓDIGO DA APLICAÇÃO MAX-MIN EM SLEDS-SD . 62
APÊNDICE C – SISTEMA DE GERAÇÃO DO INDICADOR DE POTÊNCIA DO SINAL RECEBIDO 65

1 INTRODUÇÃO

Os sistemas ciber-físicos e a Internet das Coisas dependem de dispositivos sensores para capturar dados do ambiente e dos usuários de aplicações. Eles são os principais componentes que compõem a camada de aquisição de dados de tais sistemas (Ahmed et al., 2019). Devido ao número crescente de dispositivos e ao volume de dados que eles coletam, são necessários modelos de armazenamento novos e eficientes. A tecnologia de dispositivos sensores se desenvolveu rapidamente, proporcionando-lhes maior poder de processamento e capacidade de armazenamento. Uma técnica comum, preconizada pela computação de borda (Satyanarayanan, 2017) e de névoa (Dastjerdi e Buyya, 2016), é processar e armazenar os dados em uma extremidade próxima da fonte de dados. Em consonância com esta tendência, é possível designar alguns dispositivos sensores para desempenhar o papel de repositório dos dados capturados.

A ideia de agrupar um conjunto de sensores tem sido explorada extensivamente (Abbasi e Younis, 2007), principalmente para economizar energia de dispositivos com recursos limitados. As técnicas de agrupamento propostas diferem em sua estratégia de agrupamento de sensores, como proximidade espacial e similaridade de leituras, e na escolha de seu sensor representativo. Esse sensor é geralmente chamado de *Cluster-Head* (CH) e é responsável por coletar as leituras dos membros do *cluster* e responder às solicitações de suas leituras. Em uma arquitetura de computação de borda de quatro camadas, composta por sensores de borda, dispositivos de borda, infraestrutura de borda e a nuvem centralizada, esses sensores representantes podem desempenhar o papel de dispositivos de borda (Lin et al., 2017). Neste trabalho, eles são chamados de repositórios.

Os repositórios são responsáveis por responder às requisições de leituras do sensor em tempo real e por filtrar os dados a serem encaminhados às camadas superiores da arquitetura de borda. Para explorar melhor os recursos dos dispositivos sensores, os modelos de armazenamento de redes de sensores sem fio (RSSF) devem levar em consideração os requisitos da aplicação. Assim, geralmente novas aplicações demandam o desenvolvimento de novos modelos. No entanto, o desenvolvimento de tais sistemas é uma tarefa complexa. Observa-se que poucos estudos na literatura propõem uma abordagem que trate da modelagem e implementação de sistemas de armazenamento e consulta de dados em rede de forma sistemática, que são requisitos necessários para atender à grande demanda por serviços ubíquos em larga escala (Cattani et al., 2014) (Heinis, 2019).

Na literatura são encontrados propostas para apoiar a modelagem e desenvolvimento de sistemas de armazenamento para RSSFs. Trabalhos como (Taherkordi et al., 2015) (Soukaras et al., 2015) (Lekidis et al., 2018) propõem modelos de desenvolvimento baseados em máquina de estados, considerando apenas transições por eventos. Esse modelo é amplamente utilizado para representar sistemas reativos, como as aplicações de RSSFs. Porém, especificar o fluxo de execução de sistemas de armazenamento utilizando eventos é uma tarefa difícil, visto que algumas etapas possuem apenas processamento lógico.

Para resolver este problema, nesta dissertação é proposto um modelo de desenvolvimento para sistemas de armazenamento baseados em repositório para RSSF, denominado SMDM-SD (*State Machine Development Model for Sensor Devices*). No SMDM-SD o programador primeiro define o fluxo geral de execução do sistema como uma máquina de estados com transições lógicas e dois tipos de transições baseadas em eventos: pelo recebimento de uma solicitação e pelo tempo limite de um temporizador. As transições lógicas e baseadas em eventos são necessárias e os dois tipos de transições baseadas em eventos são suficientes para desenvolver

os modelos de armazenamento para RSSFs. Com base nessa observação, foi desenvolvida uma linguagem específica de domínio (DSL), denominada SLEDS-SD. Um programa em SLEDS-SD é composto de um conjunto de estados. A linguagem oferece suporte a estruturas de controle que se assemelham a esses tipos de transições. Essa semelhança simplifica a tarefa de programação na linguagem proposta. Como resultado do nível de abstração fornecido pelo SLEDS-SD, o desenvolvedor não precisa se preocupar com detalhes de baixo nível das linguagens de programação de sensores.

No entanto, para implantar o sistema em RSSFs, é necessário gerar código para dispositivos sensores reais. Para fazer isso, foi implementado um tradutor de SLEDS-SD para um sistema operacional e linguagem de sensor de destino. Na implementação atual, é gerado código em nesC (Gay et al., 2003) para dispositivos TinyOS [1]. Um programa SLEDS-SD pode ser visto como uma orquestração, que reúne e coordena serviços fornecidos por componentes existentes ou a serem desenvolvidos. Desacoplar entidades da aplicação, que fornecem serviços como componentes de software, da especificação do fluxo de controle da aplicação em um alto nível de abstração melhora a legibilidade do código. Além disso, esta estrategia promove a reutilização do código tanto dos componentes, que podem ser orquestrados de diferentes formas, quanto do programa SLEDS-SD, que pode utilizar o mesmo fluxo de controle com diferentes implementações dos componentes.

1.1 OBJETIVOS E CONTRIBUIÇÕES

Conforme o número de dispositivos de sensoriamento cresce, bem como o número de aplicações, novos sistemas de armazenamento de dados são exigidos. Porém, o desenvolvimento de tais sistemas não é uma tarefa simples, visto que cada sistema de armazenamento de dados deve ser planejado para se adequar a um uso específico. Portanto, o objetivo geral desta dissertação é propor um modelo de desenvolvimento que aumente a produtividade na construção de sistemas de armazenamento de dados em sensores. Para isso, os seguintes objetivos específicos foram considerados:

- elaborar um modelo de desenvolvimento de sistemas de armazenamento de dados para RSSF;
- especificar uma linguagem de domínio especifico e implementar o compilador da linguagem para geração de códigos em nesC;
- integrar a linguagem ao modelo de desenvolvimento para facilitar a implementação dos componentes de aplicação;
- realizar um estudo experimental com modelos de armazenamento presentes na literatura para validação da proposta.

1.2 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está dividida em 7 capítulos, incluindo a **Introdução** como o Capítulo 1. No Capítulo 2 está a **Fundamentação Teórica**. Neste capítulo são explorados os conceitos abordados nesta pesquisa, como Redes de Sensores Sem Fio, Modelos de armazenamento para RSSF, Métodos Formais, Desenvolvimento Baseado em Componentes e Linguagem de Domínio Específico.

[1] Planejamos gerar código para outras linguagens no futuro

No Capítulo 3 estão os **Trabalhos Relacionados**. São apresentadas propostas que focam na construção de códigos nesC ou propõem metodologias de desenvolvimento para geração de código para sensores. Neste capítulo também são encontradas comparações e discussões sobre os trabalhos descritos.

Em seguida, no Capítulo 4, é apresentado o **SMDM-SD**. São apresentados o conceito da proposta e sua arquitetura. O Capítulo 5 trata da descrição da **Linguagem SLEDS-SD**. Neste capítulo encontram-se a especificação da linguagem, a construção do compilador e a geração de código de coordenação de componentes em nesC.

No Capítulo 6 são tratados os **Estudos Experimentais**. Primeiramente foram implementados três sistemas de armazenamento em RSSF distintos. Para a avaliação da proposta foram utilizadas as métricas LOC (*Lines of (source) Code* - Linhas de Código) para avaliar o número de linhas reutilizadas entre as aplicações, e a avaliação do comportamento da aplicação, para averiguar a qualidade do código gerado pela linguagem SLEDS-SD. Por fim, o Capítulo 7 apresenta as **Considerações Finais** e os trabalhos futuros desta pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é descrita a base teórica utilizada nesta dissertação. São descritos conceitos relacionados às redes de sensores sem fio (Seção 2.1), modelos de armazenamento em RSSF (Seção 2.2), especificação de aplicações para sensores (Seção 2.3), e linguagens de desenvolvimento para sensores (Seção 2.4). No final deste capítulo, na Seção 2.5, encontra-se um resumo dos conceitos abordados.

2.1 REDES DE SENSORES SEM FIO

As Redes de Sensores Sem Fio (RSSF) são importantes componentes da Internet das Coisas (Khan et al., 2015). Elas são muito utilizadas na observação e controle de ambientes, responsáveis por monitorar, transmitir e processar dados do mundo real (Costa, 2011). Uma das finalidade das RSSFs é o monitoramento de fenômenos em locais perigosos ou de difícil acesso. Elas são aplicadas também em serviços militares, centros de saúde, indústrias e vigilância domiciliar (Singh et al., 2015). A recarga ou substituição da fonte de energia dos dispositivos sensores, normalmente uma bateria, algumas vezes é financeiramente ou logisticamente inviável. Apesar dessa restrição energética, é desejável que a vida útil da RSSF seja a mais longa possível (Shelke et al., 2013).

As RSSFs são um ramo das redes **Ad hoc**, que designam para o papel de nó *sink* (coletor) um ou mais sensores ou algum dispositivo de maior capacidade, chamado de estação base. Esse nó é a conexão entre a rede e um ambiente externo. Os nós sensores detectam as mudanças no ambiente e transmitem as informações por métodos variados para o nó sink. A Figura 2.1 ilustra exemplos de topologias arquitetônicas na perspectiva das RSSF.

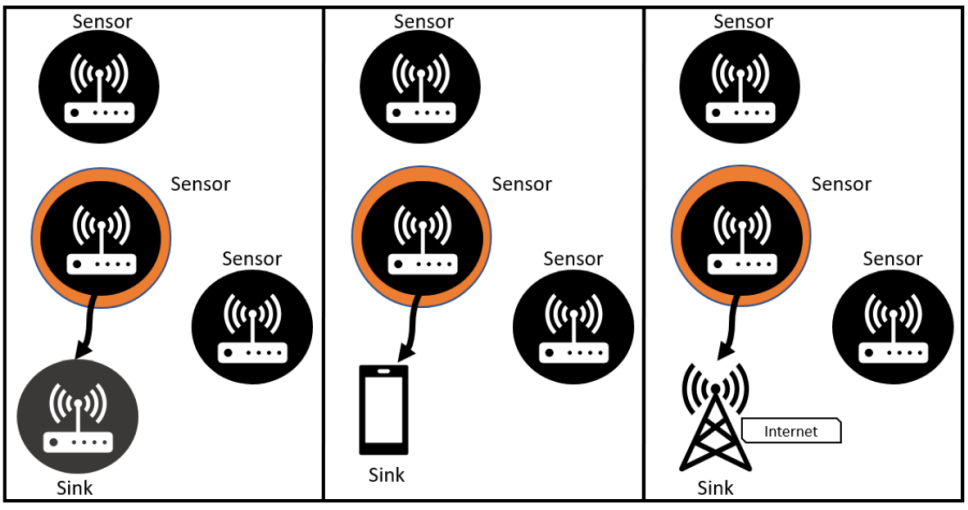


Figura 2.1: Topologias arquitetônicas na perspectiva das RSSFs. Adaptado de (Rashid e Rehmani, 2016).

Na Figura 2.1 à esquerda, um sensor pode receber o papel de *sink*, sendo o responsável por centralizar as leituras obtidas na rede. No exemplo do meio, um aparelho coletor, como um PDA (*Personal digital assistant*) ou smartphone, é conectado à rede para receber as leituras. Finalmente, mais à direita, um ponto de acesso a Internet é utilizando como *sink*. Apesar da simplicidade na coleta das leituras da rede por meio do *sink*, o roteamento dos dados até este

ponto de acesso precisa ser bem planejado, pois a maneira como os sensores transmitem suas leituras pode afetar o tempo de vida da rede. Por exemplo, se os sensores enviarem suas leituras diretamente para o *sink*, a distância entre ambos pode exigir que o sensor gaste mais energia para transmitir o sinal de frequência de rádio (Radio Frequency – RF) em um raio maior. Da mesma forma que, se a rede utilizar o roteamento dos dados entre seus vizinhos, os nós mais próximos do *sink* ficam sobrecarregados e consequentemente gastarão mais energia. Na seção 2.2 é feita uma classificação dos modelos de armazenamento e roteamento de dados.

Os componentes de uma RSSF são o sensor, o fenômeno e o observador. O **sensor** é o dispositivo responsável pelo monitoramento de uma determinada caraterística de um ambiente. Essa característica monitorada trata-se do **fenômeno**. Já o **observador** está diretamente relacionado à finalidade da RSSF, sendo o responsável por compreender/utilizar as leituras obtidas.

2.1.1 Hardware

Os dispositivos de sensoriamento possuem dimensões e recursos limitados. Normalmente, a estrutura básica desse tipo de dispositivo é formada por: transceptor (rádio), processador, memória, sensores e bateria (Figura 2.2). Porém, cada nó sensor pode ser adaptado de acordo com o ambiente. Por exemplo, se um sensor for instalado no subsolo, ele terá transceptores com alta potência de transmissão para superar atenuações de canais ruidosos (Rashid e Rehmani, 2016).

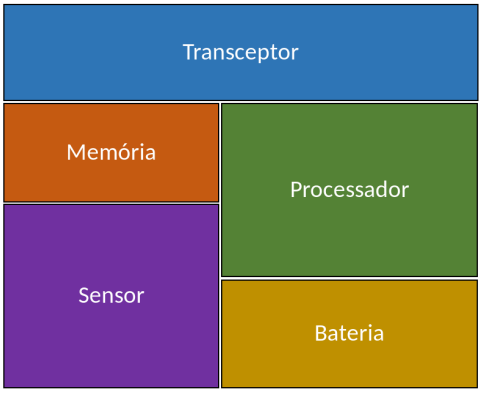


Figura 2.2: Componentes de um nó sensor. Adaptado de (Ruiz, 2003).

O **transceptor**, também conhecido como rádio, é o responsável pela troca de mensagens entre os nós. Ele é amplamente utilizado nas RSSFs pois as transmissões de radio-frequência podem percorrer longas distâncias e não possuem limitações físicas como cabeamento. A **memória** da maioria dos dispositivos é limitada, tornando inviável o armazenamento de todos os dados capturados localmente. Semelhante à memória, o **processador** dos dispositivos não foi desenvolvido para realizar computações complexas devido às restrições do *hardware*, como dimensões e consumo de energia.

O **sensor** é um componente do hardware que varia de acordo com a finalidade da rede. Os sensores mais conhecidos são os de luminosidade, temperatura, umidade relativa, pressão barométrica, aceleração, sísmico, acústico e magnético. Por fim, a **bateria** é um componente com a qual a maioria das aplicações em RSSF se preocupam. Como a bateria é um recurso escasso, ele limita todos os demais. Por exemplo, o transceptor deve conter uma alta eficiência energética, sendo ligado apenas em períodos pré-determinados.

2.1.2 Sistemas Operacionais

Os sistemas operacionais (SO) para RSSFs são limitados, devido às restrições impostas pelo *hardware* dos dispositivos. Assim, o desenvolvimento de sistemas torna-se complexo, pois a linguagem de programação de cada SO também possui suas limitações específicas. Dentre os SOs utilizados em RSSF podem ser citados: TinyOS, Contiki e ScatterWeb. Além disso , dentre os SOs com foco em IoT, estão o Android e o Middleware J2SE.

O SO embarcado de código aberto **TinyOS**, desenvolvido pela Universidade da Califórnia em Berkeley em parceria com a Intel, foi implementado para atender RSSFs de diversas escalas. Ele tem como objetivo atender à demanda por SOs otimizados voltados aos dispositivos com poucos recursos de armazenamento e processamento (Hill et al., 2000). Por exemplo, o TinyOS, entre suas principais vantagens, ocupa pouco espaço de memória. O TinyOS utiliza uma arquitetura baseada em componentes e tem compatibilidade com dezenas de microcontroladores.

O **Contiki** (Dunkels et al., 2004) é um SO de código aberto para pequenas redes de sensores, desenvolvido pelo Instituto Sueco de Ciência da Computação. Ele foi implementado na linguagem C, sendo portável a varias arquiteturas de microcontroladores, como o MSP430 e o Atmel AVR (Myklebust, 1996). O Contiki foi projetado para atender sensores com memória limitada. Ele possui uma camada de compressão do protocolo de rede IPv6, o 6LoWPAN, permitindo aos sensores utilizar o UDP (*User Datagram Protocol*).

O **Scatter Web**[1] é um SO proposto pelo Departamento de Matemática e Ciência da Computação da Universidade Livre de Berlim, para sensores auto-configuráveis (Schiller et al., 2005). O ScatterWeb é uma plataforma heterogênea distribuída para a implantação de redes de sensores. Ele possui código aberto e foi desenvolvido para o *hardware* ScatterWeb, desenvolvido pelo mesmo grupo de pesquisa.

Com relação à IoT, o SO **Android** foi desenvolvido pela Google para oferecer suporte a dispositivos móveis, como *smartphones*, *tablets* e relógios inteligentes. Os dispositivos com Android não se preocupam com restrições energéticas e de armazenamento. Assim, o processamento e armazenamento local tornam-se viáveis. Normalmente os dispositivos possuem diversos sensores, como acelerômetro, giroscópio, magnetômetro, GPS, barômetro, luminosidade, entre outros.

O ***middleware* J2SE** foi desenvolvido para ser uma plataforma que fornece serviços necessários para a execução de aplicações Java em dispositivos. Essa plataforma é muito utilizada na IoT em eletrodomésticos inteligentes, como geladeiras, cafeteiras e ar-condicionado.

Resumindo, seja para RSSF ou para a IoT, cada SO para dispositivos possui suas particularidades. Quando voltado às RSSF, é necessário focar em economia de recursos. Já na IoT, o foco principal é atender o heterogeneidade dos dispositivos, das aplicações e dos tipos de sensores.

2.2 MODELOS DE ARMAZENAMENTO EM RSSF

Devido às restrições mencionadas anteriormente, as RSSFs necessitam de modelos de armazenamento eficientes. Cada modelo possui uma finalidade específica, que atende a um determinado tipo de aplicação da rede. De acordo com Carrero (2018), os modelos de armazenamento para RSSFs podem ser classificados em externo e na rede, sendo que o armazenamento na rede pode ser em repositórios (centrado), ou local.

As redes de sensores possuem nós especiais que se diferenciam dos sensores convencionais, como o *sink* e o *cluster-head* (líder). Nós *sink* são responsáveis por centralizar as leituras de

[1] `https://www.mi.fu-berlin.de/inf/groups/ag-tech/projects/ScatterWeb`

todos os nós da rede para repassá-las para o usuário. Quanto ao *cluster-head*, ele monitora e coordena agrupamentos de nós escravos presentes na rede.

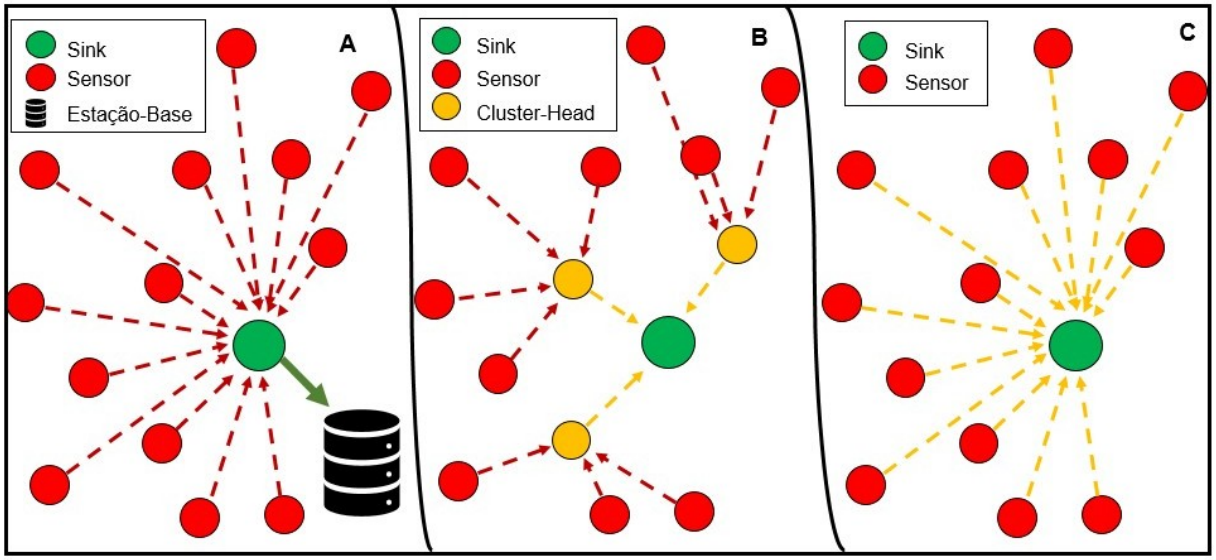


Figura 2.3: Classificação dos modelos de armazenamento em RSSF.

A Figura 2.3 (A) ilustra o armazenamento externo, no qual os dados sensoreados são enviados a um equipamento externo à rede, chamado de estação base (EB). Ela é o componente desse modelo de armazenamento que possui maior poder de armazenamento e processamento. Nesse modelo de armazenamento, os nós sempre enviam suas leituras para a EB.

Diferentemente, o armazenamento na rede foca no armazenamento dos dados sensoreados próximo à borda. O armazenamento na rede em repositórios, ilustrado na Figura 2.3 (B), considera a criação de grupos de sensores que transmitem suas leituras para um líder de grupo (*Cluster-Head* - CH), que armazena e retransmite as leituras do agrupamento para o *Sink* quando necessário. Já no armazenamento na rede local, apresentado na Figura 2.3 (C), os próprios sensores armazenam suas leituras e as enviam para o *Sink* quando necessário.

O modelo de armazenamento em repositórios possui algumas vantagens que o fazem se sobressair aos outros modelos, como por exemplo, a alta escalabilidade e economia energética. Os modelos de armazenamento em repositórios também se mostram eficientes no quesito roteamento. Na Figura 2.4, os sensores A, B e C estão transmitindo suas leituras para o nó *sink* S.

Na Figura 2.4 (2), os sensores em destaque desempenham o papel de CH, servindo como repositórios na rede. O primeiro sensor em destaque armazena e realiza o roteamento dos dados dos sensores A e B agregados em ab. Já o segundo sensor em destaque recebe a transmissão ab e agrega com os dados que mantém armazenado, no caso c. No roteamento tradicional foram necessárias 9 transmissões, enquanto que, no roteamento centrado com agregação de dados foram necessárias apenas 6 transmissões. Assim, esse sensor envia a mensagem abc para o *sink*. Com esse modelo é possível reduzir o tráfego e economizar energia em RSSF.

2.3 ESPECIFICAÇÃO DE APLICAÇÕES PARA SENSORES

A especificação de sistemas para RSSF é complexa, devido à natureza reativa desse tipo de rede. Dada essa dificuldade, a utilização de um *método formal* no desenvolvimento pode facilitar esta tarefa. Os métodos formais são formalismos matemáticos utilizados na especificação,

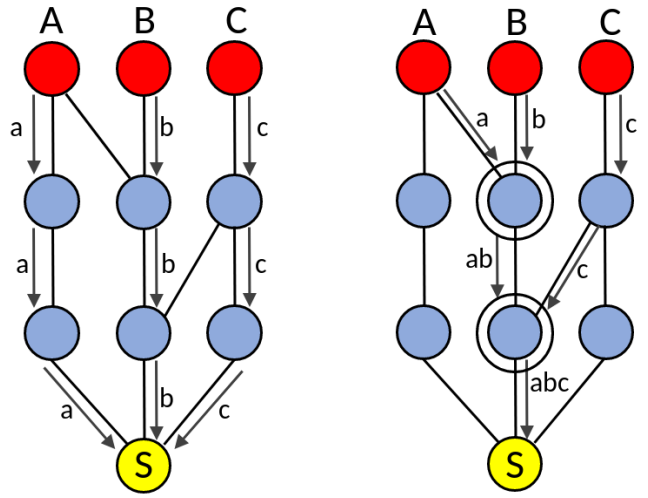


Figura 2.4: Roteamento tradicional e roteamento centrado em dados. Adaptado de (Loureiro et al., 2003).

desenvolvimento e avaliação de *software*. Cada método formal contém características únicas que o restringem a modelos de sistema específicos. Dessa forma, é necessário compreender o método formal que mais se ajusta ao sistema proposto.

As RSSF estão em uma contínua interação com o ambiente monitorado, sendo acionadas no ritmo determinado pelo ambiente (Romadi e Berbia, 2008). Nesse contexto, a especificação, desenvolvimento e validação de sistemas em RSSF são tarefas complexas. Dessa maneira, a adoção de métodos formais que representem a natureza das RSSF é importante para o desenvolvimento de sistemas de armazenamento confiáveis. Nos trabalhos atuais nota-se que os métodos formais (i) máquina de estados, (ii) componentes de software, e (iii) meta-modelos são constantemente utilizados na especificação dos sistemas e na construção dos modelo de desenvolvimento. Suas características são descritas a seguir.

2.3.1 Máquina de Estados

A máquina de estados (ME) é muito utilizada para representar sistemas de interface para o usuário e circuitos lógicos. A ME contém três componentes essenciais: o estado, a ação e a transição. Um **estado** é a representação de uma fase de execução do objeto estudado. Em um estado estão uma série de ações que definem a próxima transição. A **ação** é uma atividade a ser realizada. As ações são a parte lógica da aplicação, sendo responsáveis por realizar um determinado processamento. Uma **transição** é uma mudança de estado. Nesse formalismo, o estado que está sendo executado é o *estado atual*, sendo que só pode haver um estado atual por vez. Ele é apropriado para o desenvolvimento em RSSFs, pois é capaz de expressar de maneira simples o comportamento lógico e reativo desse tipo de sistema, sendo muito utilizado na representação do fluxo de execução.

Na Figura 2.5 é ilustrado a representação de um sistema de monitoramento que toca um alarme de acordo com o acionamento do sensor. O sensor permanece em *stand-by* até ser acionado, e então é realizada a transição para o estado sensor acionado. Nesse estado, se o sensor permaneceu acionado por 20 segundos, é realizada a mudança para o estado tocar alarme. Porém, se o sensor passou 5 segundos desativado, a transição é para o estado inicial (desligado).

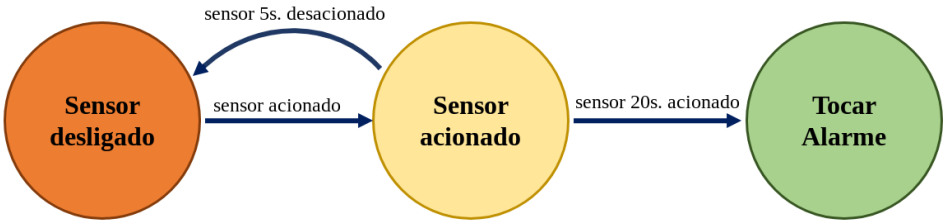


Figura 2.5: Máquina de estados para representação do fluxo de execução de um sistema.

2.3.2 Componentes

O método formal de componentes é um formalismo que propõe o particionamento de sistemas em unidades, além de promover o uso de componentes de software pré-existentes no desenvolvimento (Lau e di Cola, 2017). Entre as principais vantagens da utilização do modelo de desenvolvimento baseado em componentes estão: a **produtividade**, pois é possível economizar tempo reaproveitando componentes já desenvolvidos, o **padrão de desenvolvimento**, e a **robustez**. A maior qualidade do produto final está relacionada à separação de funcionalidades específicas de código.

Em RSSFs a programação baseada em componentes foi reconhecida como uma abordagem eficaz para atender requisitos importantes como facilidade de programação, boa-estruturação do código, grau de reutilização, esforço de desenvolvimento de software necessário e a capacidade de ajustar o software do sensor para uma aplicação específica (Taherkordi et al., 2011).

Além dos sistemas para RSSF, a abordagem de desenvolvimento baseada em componentes é adotada em diversos domínios de aplicações (Chaudron et al., 2005). A ideia básica do desenvolvimento baseado em componentes, ilustrada na Figura 2.6, é desenvolver um repositório de componentes que são reutilizados por diversas aplicações.

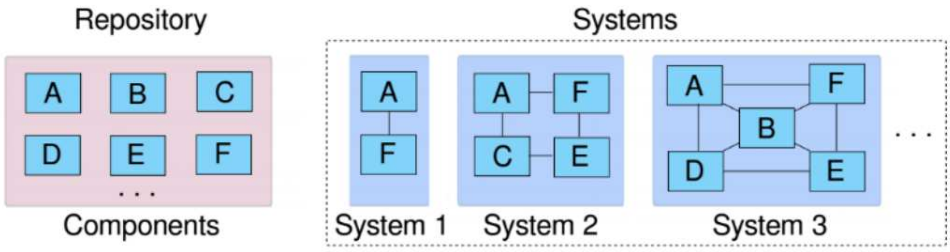


Figura 2.6: Desenvolvimento de software baseado em componentes. (Lau e di Cola, 2017)

Um componente é compreendido como parte de uma composição. Na computação, um componente pode ser um bloco de código, um objeto ou uma aplicação. Sendo assim, normalmente um componente é parte de um sistema como uma funcionalidade lógica. Os componentes proporcionam o particionamento dos sistemas em subsistemas ou funcionalidades, auxiliando na modularização das aplicações. A utilização de interfaces possibilita a realização de chamadas aos componentes. O código específico das funcionalidade é separado do código principal da aplicação, diminuindo a complexidade do desenvolvimento.

2.3.3 Metamodelos

Metamodelo é um formalismo que propõe o uso de uma estrutura pré-definida (os modelos), para o desenvolvimento de sistemas. No desenvolvimento utilizando métodos formais, um processo essencial é a modelagem. A modelagem é responsável por explicar as características ou comportamento de um sistema. Assim, a utilização de metamodelos exige que o processo de modelagem especifique as particularidades da entidade tratada. Um metamodelo é um modelo que define a estrutura, a semântica e as limitações dos modelos. Por exemplo, os modelos UML (Linguagem de Modelagem Unificada) são definidos pelo Metamodelo UML.

O Desenvolvimento Orientado a Modelo (em inglês, *Model-Driven Development* - MDD) é uma técnica que visa a criação de uma classe de modelos principal (Vara e Marcos, 2012) que se ramifica em outros sub-modelos. Dependendo do nível de abstração, essa técnica de desenvolvimento permite que aplicações inteiras possam ser reutilizadas em diferentes SOs. Na figura 2.7 são ilustrados os principais elementos de um MDD.

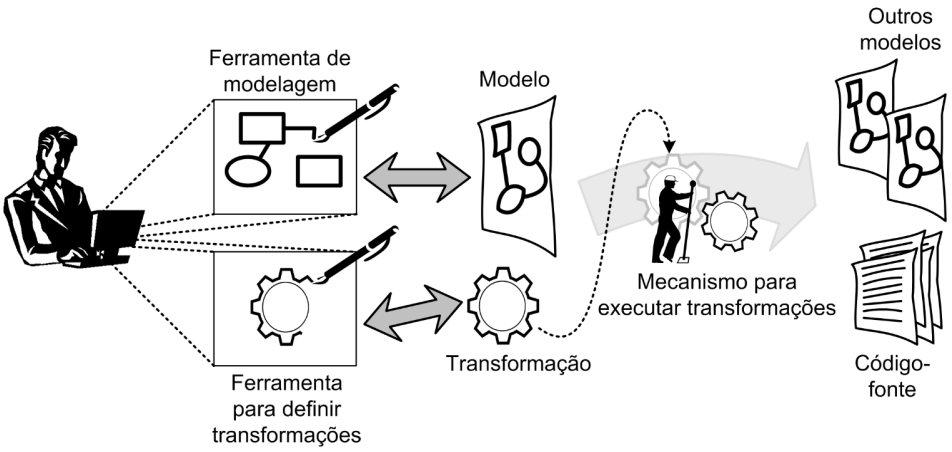


Figura 2.7: Principais Elementos do MDD. (Lucrédio, 2009)

Inicialmente, o desenvolvedor utiliza uma ferramenta para realizar a modelagem do sistema. Também é necessária uma ferramenta para definir as transformações dos modelos para o código-alvo. Com a conclusão do desenvolvimento do modelo, somado à ferramenta de transformação, são utilizados mecanismos para a transformação do modelo em outros modelos ou em código-fonte.

Concluindo, os modelos proporcionam um nível elevado de abstração de código, alguns chegando a ser independente de plataforma. Entre as principais vantagens de utilizar essa técnica é a possibilidade de implementar sistemas independentes do SO. Porém, as regras definidas pela estrutura do modelo podem limitar alguns recursos. Por exemplo, um metamodelo que gera código para sensores independente de plataforma, como o Wiselib (Baumgartner et al., 2010), precisa se preocupar com as limitações de todos os ambientes para que o modelo seja genérico.

2.4 LINGUAGENS DE DESENVOLVIMENTO PARA SENSORES

As linguagens de desenvolvimento para sensores foram criadas de acordo com as limitações dos SO que elas atendem. Por exemplo a linguagem nesC, desenvolvida para implementar aplicações no TinyOS, utiliza a arquitetura baseada em componentes, semelhante ao SO. Nessa seção são descritas as particularidades das linguagens de desenvolvimento para

sensores, como nesC, C, Java e OTcl. Além disso, é descrito o conceito de linguagens de domínio especifico, bem como a construção e o funcionamento de compiladores para a linguagem.

Linguagem nesC. Para o desenvolvimento de sistemas para o TinyOS foi proposto o **nesC**, uma linguagem baseada em componentes e orientada a eventos (Gay et al., 2014). Semelhante à linguagem utilizada no Contiki, o nesC também é uma adaptação da linguagem C para dispositivos com poucos recursos. Os componentes são "amarrados" por meio de arquivos de configuração, e as chamadas de funcionalidades dos componentes são realizadas por meio das interfaces. A implementação dos componentes é realizada no aquivo de módulo. Os componentes do nesC focam na abstração do hardware, porém é possível implementar funcionalidades lógicas nos mesmos. Esse modelo de programação funciona bem para aplicações nas quais não há complexidade na definição do fluxo de execução. Por exemplo, o Listing 2.1 é a implementação do módulo da aplicação Blink. Nessa implementação, o LED vermelho alterna periodicamente.

```
1  module BlinkC @safe() {
2    uses interface Timer<TMilli> as Timer0;
3    uses interface Leds;
4    uses interface Boot;
5  }
6  implementation {
7      event void Boot.booted() {
8          call Timer0. startPeriodic ( 250 ); }
9
10     event void Timer0. fired () {
11         call Leds.led0Toggle (); } }
```

Listagem 2.1: Implementação da aplicação Blink

Primeiramente é necessário realizar a importação da interface dos componentes que são utilizados na aplicação. Nas linhas 2, 3 e 4 são importados os componentes *Timer, Leds* e *Boot*. O `Timer` é o componente que gerencia os temporizadores da aplicação. Nesse componente é possível acionar temporizadores que disparam periodicamente ou uma única vez (*one shot*). O componente `Leds`, como o próprio nome já diz, é responsável por gerenciar os leds do dispositivo, com funcionalidades como *ON* (ligar), *OFF* (desligar) e *TOGGLE* (Alternar). Por fim, o componente `Boot` tem a função iniciar e encerrar a aplicação. Na seção de implementação (linha 6 - 11) é realizada a execução da aplicação, bem como as chamadas de funcionalidades dos componentes. A parte lógica da aplicação fica alojada em eventos, como no evento *Boot.booted* que, ao ser acionado, realiza uma chamada da funcionalidade *Timer0.startPeriodic* passando como parâmetro um valor em milissegundos. Outro evento é o disparo do temporizador, o *Timer0.fired* (linha 10). Esse evento acontece no final de cada período de tempo. Portanto, encerrando a lógica da aplicação, periodicamente um temporizador é inicializado, e ao seu final o *led* vermelho acende ou apaga por meio da funcionalidade *Leds.led0Toggle*.

Linguagem C. No desenvolvimento de sistemas para o SO Contiki, é utilizada uma adaptação da linguagem C capaz de executar processos em paralelo. A programação nessa linguagem é baseada em processos e eventos, semelhante ao nesC. Para inciar uma aplicação, o desenvolvedor deve implementar uma estrutura (como uma lista) para gerenciar o fluxo de execução dos processos. Cada processo contém funcionalidades específicas no nível de hardware, como o comando de acender os Leds ou de iniciar um cronômetro.

Linguagem Java. A linguagem Java é muito utilizada em dispositivos embarcados, como os utilizados na IoT (Chen, 2001). Atualmente 3 bilhões de dispositivos executam o Java [2]. O Java é uma linguagem que utiliza o paradigma de Programação Orientada a Objetos (POO).

[2] `https://www.oracle.com/br/java/`

Neste paradigma, o desenvolvedor consegue abstrair os objetos do sistema, bem como suas funcionalidades. Para desenvolvimento desenvolvimento na plataforma J2SE, a linguagem Java realiza a importação das bibliotecas que executam as funcionalidade de hardware necessárias.

Linguagem Otcl. O Otcl (*Tool Command Language OO*) é a linguagem de desenvolvimento de sistemas para o simulador NS-2 (*Network Simulator - 2*). Essa linguagem, semelhante ao Java, utiliza o paradigma de POO. Assim, é possível identificar as características dos objetos, bem como suas funcionalidades, e abstrair essa parte do código para uma classe especifica.

2.4.1 Linguagem de Domínio Específico

Uma Linguagem de Domínio Específico (*Domain-Specific Language* - DSL, em inglês) é um paradigma que aproxima o desenvolvimento do software do domínio da aplicação. "Uma DSL fornece ao especialista em domínio termos e notações que correspondem ao seu espaço cognitivo e intuição"(Sadilek, 2007). Assim, é possível compreender que as DSLs tornam o desenvolvimento de um domínio específico mais simples do que as linguagens de uso geral.

A construção de uma DSL ocorre por meio da especificação do comportamento dos sistemas em um determinado domínio. A DSL é utilizada para elevar o nível de abstração de linguagens de programação próximas à máquina, como o *assembly* ou o C. Assim, apesar de trabalhoso, a construção de uma DSL aumenta o nível de produtividade no desenvolvimento dos sistemas do determinado domínio. O uso de um método formal na especificação da DSL garante a integridade da aplicação e auxilia o desenvolvedor na implementação da especificação.

Após a especificação do domínio da linguagem, é necessário o desenvolvimento de uma ferramenta que interprete a gramática proposta e realize a tradução da mesma para uma linguagem específica. Essa ferramenta trata-se do compilador. A seguir são abordados os conceitos de compiladores, bem como as etapas necessárias para a tradução.

2.4.1.1 Compiladores

Um compilador é um programa que recebe uma **linguagem fonte** de entrada, e a traduz para uma **linguagem alvo** (Aho et al., 2007). Um exemplo é o compilador da LP (Linguagem de Programação) C, que recebe o código em C (uma linguagem próxima da humana) como entrada, e gera um código de baixo nível como saída, em linguagem de máquina. Louden e Silva (2004) dividem o processo de tradução de um compilador nos módulos (i) Analisador Léxico, (ii) Analisador Sintático, (iii) Analisador Semântico e (iv) Gerador de Código. A ferramenta Flex é uma das mais utilizadas para análise léxica, e em conjunto com a ferramenta Bison, pode implementar um compilador completo, visto que o Bison trata das análises sintática e semântica, além da geração de código (Aho et al., 2007). Para exemplificar as etapas da compilação é considerada uma expressão que calcula um valor final (montante), de acordo com o valor inicial (depósito inicial) e os juros (taxa de juros).

montante:= deposito_inicial + taxa_de_juros * 60

A **Análise léxica** é responsável por identificar as palavras reservadas da DSL e retornar os valores para a análise semântica em formato de *tokens*. Esses *tokens* são enviados sequencialmente como parâmetros de entrada para a próxima fase da compilação. Essa etapa está diretamente relacionada com a gramática da linguagem. De acordo com a expressão apresentada anteriormente, foram identificados os seguintes *tokens*:

- Os identificadores: montante, deposito_inicial, taxa_de_juros

- Os símbolos de atribuição, adição e multiplicação
- O número 60

Na etapa de **Análise sintática** é gerada a AST (*Abstract Syntax Tree*), ilustrada na Figura 2.8. Nessa estrutura, os *tokens* são organizados sequencialmente formando sentenças gramaticais que são utilizadas pelo compilador. Essas sentenças são formadas de acordo com as regras da linguagem. Por exemplo, devido à precedência do sinal de multiplicação sobre o sinal de adição na AST é necessário separar essas operações em duas sentenças.

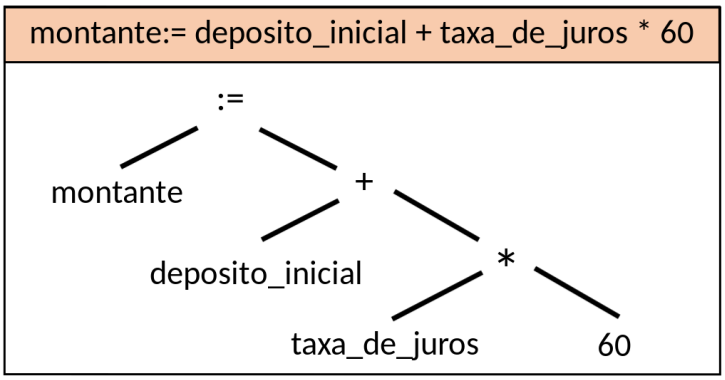


Figura 2.8: Exemplo de árvore sintática. Adaptado de (Aho et al., 1998).

Já a **Análise semântica** verifica se as expressões construídas na AST por meio de sequências de *tokens* tem sentido semântico. Essa verificação é realizada por meio das regras da gramática. Além disso, essa etapa realiza a verificação de tipos. Assim, a análise semântica é responsável por capturar informações de tipos, bem como retornar erros semânticos.

Finalmente, a **Geração de código** realiza a tradução da linguagem fonte para a linguagem alvo por meio da representação da árvore sintática. Para isso, **tradução dirigida pela sintaxe** é um procedimento que consiste na geração do código em tempo de compilação. Para isso, para cada regra com sentido semântico, é agregada a estrutura do código alvo. Assim, no final de cada regra o código correspondente é gerado.

2.5 CONSIDERAÇÕES FINAIS

Neste capítulo foram descritos os conceitos explorados nesta dissertação. Primeiramente, na seção 2.1 foram abordadas as definições das RSSF, bem como os seus componentes de hardware e software. Vale destacar que as RSSF são uma tecnologia cada vez mais presente no cotidiano das pessoas, mesmo que seja de maneira quase imperceptível. Elas são responsáveis por captar dados do ambiente. Assim, seu hardware, os sensores, são preparados para cada fim especifico da rede. Por exemplo, se é uma RSSF que realiza um monitoramento no mar, os sensores devem ser à prova d'água. Os sensores também possuem restrições, como de bateria, armazenamento e processamento. Assim, as aplicações, bem como os SO para RSSF, devem ser otimizados, evitando o desperdício dos recursos. Ainda nessa seção foi abordado o SO TinyOS e sua linguagem, o nesC. É exposto que o TinyOS é eficiente para as RSSFs com poucos recursos. Quanto à linguagem nesC, mesmo sendo baseada em componentes orientados a eventos, ela não tem uma abordagem de desenvolvimento que auxilie o desenvolvedor na definição dos componentes e especificação do fluxo de execução.

Na seção 2.2 foram descritos os modelos de armazenamento em sensores. Estes modelos são classificados em armazenamento externo e armazenamento na rede. No armazenamento externo, os dados sensoreados são diretamente enviados para um local fora da rede, como uma EB. Já no armazenamento na rede, os dados sensoreados podem ser armazenados localmente ou em repositórios. No armazenamento local, o sensor armazena a leitura na sua própria memória. Por fim, no armazenamento em repositórios, alguns sensores fazem o papel de líderes de agrupamentos e recebem leituras dos integrantes do grupo. É notório que o armazenamento em repositório se destaca devido à alta escalabilidade proporcionada e à economia de bateria com a diminuição de transmissões.

Na seção 2.3 foram detalhadas técnicas para a especificação de aplicações em RSSF. Dada a dificuldade encontrada no desenvolvimento desses sistemas, é comum utilizar métodos formais na etapa de especificação e implementação dos sistemas. Um método formal é um formalismo matemático que pode ser utilizado na especificação, no desenvolvimento e na avaliação de software. Entre os métodos formais mais conhecidos, o de máquina de estados se assemelha à natureza reativa das RSSF. Na máquina de estados, o estado representa o estágio atual do processamento, sendo composto por um conjunto de ações. As ações são atividades que devem ser realizadas. A transição, por sua vez é uma mudança de estado por meio da ocorrência de um evento ou processamento. Nessa seção também foi descrito o método formal de componentes. Um componente pode ser um objeto, um bloco de código ou uma sub-aplicação. Entre as principais vantagens de se utilizar esse modelo de desenvolvimento estão a possibilidade de reaproveitamento de código do componente, bem como do código de execução da aplicação, e o aumento na produtividade no desenvolvimento, pois possibilita a combinação de componentes previamente existentes. Por fim, a especificação baseada em meta-modelos auxilia o desenvolvedor a reaproveitar os modelos existentes, pois esse método formal permite o desenvolvimento de código independente de plataforma.

Na seção 2.4 foram descritas as Linguagens de desenvolvimento para sensores. Foram consideradas linguagens de desenvolvimento mais comuns no desenvolvimento para sensores, como o nesC, o C, o Java e o Otcl. Nessa seção também foi abordado o conceito de DSL. Uma DSL é uma linguagem de programação desenvolvida a partir de um domínio especifico, com o objetivo de elevar o nível de abstração, bem como a produtividade no desenvolvimento. Um compilador trata de realizar a tradução de uma linguagem de alto nível para uma linguagem resultante de mais baixo nível. As etapas da compilação de uma linguagem também foram apresentadas.

3 TRABALHOS RELACIONADOS

Neste capítulo são descritos os trabalhos correlatos. Para selecionar as propostas presentes na literatura foram utilizados os seguintes critérios: (i) trabalhos que propõem modelos de desenvolvimento de sistemas para RSSF e (ii) linguagens de alto nível para abstração de códigos de sensores. Dessa maneira, é possível comparar separadamente os objetivos desta dissertação, o SMDM-SD e o SLEDS-SD, com os trabalhos que possuem objetivos similares. Um breve resumo e uma tabela comparativa entre os trabalhos são apresentados na seção 3.8.

3.1 SENNET

A DSL SenNet (Salman e Al-Yasiri, 2016) é uma linguagem que visa auxiliar o programador no desenvolvimento de aplicações para RSSF. As principais características da linguagem são a abordagem de desenvolvimento utilizando MDD (*Model-Driven Development*, em português Desenvolvimento Orientado a Modelos). Ele é baseado em redes hierárquicas e tem como objetivo a redução de detalhes complexos da programação, proporcionando alto nível de abstração, e a geração de código nesC para a implantação da aplicação em dispositivos sensores. A Figura 3.1 ilustra os estágios de desenvolvimento e geração de código de aplicações para RSSFs utilizando o SenNet.

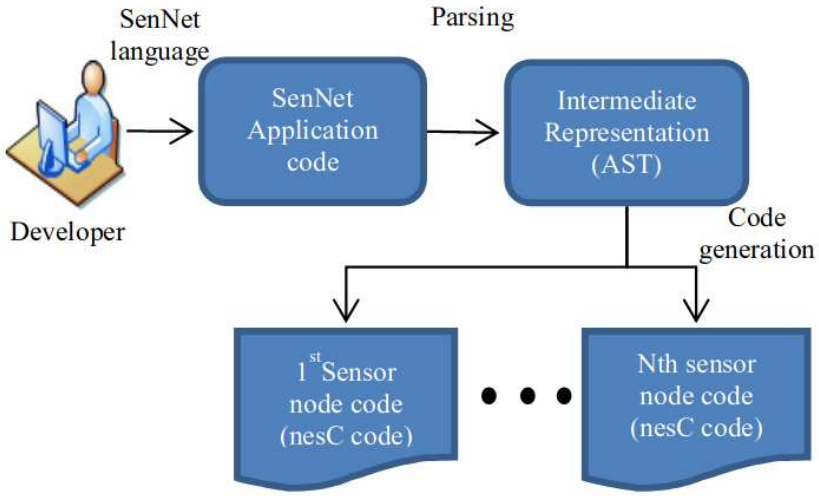


Figura 3.1: Etapas no desenvolvimento de uma aplicação utilizando SenNet. (Salman e Al-Yasiri, 2016)

A primeira etapa é a codificação da aplicação em SenNet para cada nível hierárquico utilizado na rede. Utilizando uma AST, é gerada a representação intermediária da lógica da aplicação codificada. Finalmente, utilizando a representação lógica da aplicação, a geração do código em nesC é realizada. O código nesC é gerado de acordo com as configurações da rede, informadas na preparação do ambiente de codificação.

A programação ocorre de acordo com o meta-modelo do SenNet, que possui dois componentes: configuração da rede e configuração da aplicação. O meta-modelo de configuração da rede serve para gerar código de acordo com os níveis de rede da aplicação, os quais podem ser: (a) nível de nó único, (b) nível de *cluster* ou (c) nível de rede que possa ser programada.

Dessa maneira, cada nó pode receber uma assinatura para conter uma função especial na rede, como *SensorNode, ClusterHeadNode, SinkNode ou ComputationNode*. Já o meta-modelo de configuração da aplicação contém os parâmetros da aplicação de acordo com o papel do dispositivo na rede. Também é onde o desenvolvedor programa o código específico da aplicação.

3.2 TOKENIT

O *framework* Tokenit (Taherkordi et al., 2015) oferece uma abordagem orientada a máquina de estados para a modelagem do fluxo de execução de aplicações para sensores. Na versão atual, o Tokenit gera código para o SO Contiki. Nessa proposta, os estados, denominados *tokens*, são compostos por um conjunto de atividades. A transição entre os estados é acionada por um *timer* ou por eventos assíncronos, como uma detecção de movimento.

Essa abordagem reduz a complexidade do desenvolvimento desde que, por meio da separação do fluxo de execução da aplicação do código específico de cada *token*, o programador consegue se concentrar em partes distintas da construção da aplicação. Na Figura 3.2 são ilustrados os processos do desenvolvimento utilizando o *framework* Tokenit.

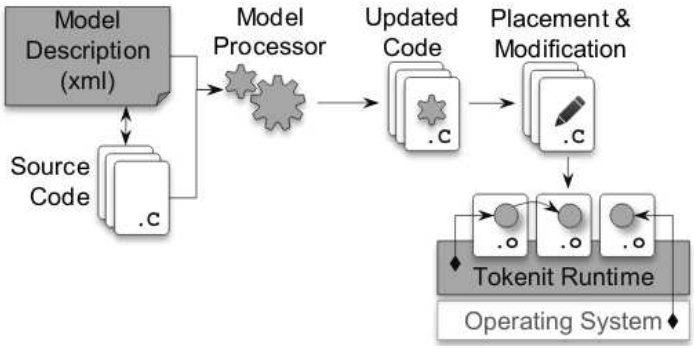


Figura 3.2: Fases de desenvolvimento utilizando o Tokenit. (Taherkordi et al., 2015)

O programador precisa desenvolver as atividades e preparar a descrição da máquina de estados no formato XML. O processador de modelo realiza a análise do modelo descrito e do código fonte de entrada, e posteriormente a geração do código. Em seguida, o programador analisa o código gerado e implementa as modificações necessárias, para posteriormente realizar a implantação da aplicação utilizando o *Tokenit Runtime*.

3.3 X-MACHINE

O *X-Machine* (Braga, 2012) é uma plataforma de desenvolvimento de aplicações em RSSFs. Essa ferramenta utiliza o método formal CMX (*Communicating X-Machine*). Nesse método formal os estados se comunicam de maneira reativa, ou seja, em resposta a algum evento. O *X-Machine* possui uma interface gráfica, incorporada à IDE Eclipse por meio de *plugins*, para desenvolvimento e geração do código das aplicações na linguagem nesC. Na Figura 3.3 é ilustrado o ambiente de desenvolvimento de aplicações utilizando o Eclipse.

O desenvolvimento de aplicações no *X-Machine* ocorre por meio da interface gráfica. Esse ambiente é composto pelo **Explorer** (A), que auxilia na navegação entre os arquivos, pelo **Modeling view** (B), janela na qual a aplicação é modelada, pela **Paleta** (C), que contém os componentes que são utilizados na modelagem, pelo **Propertiew View**, que permite ao

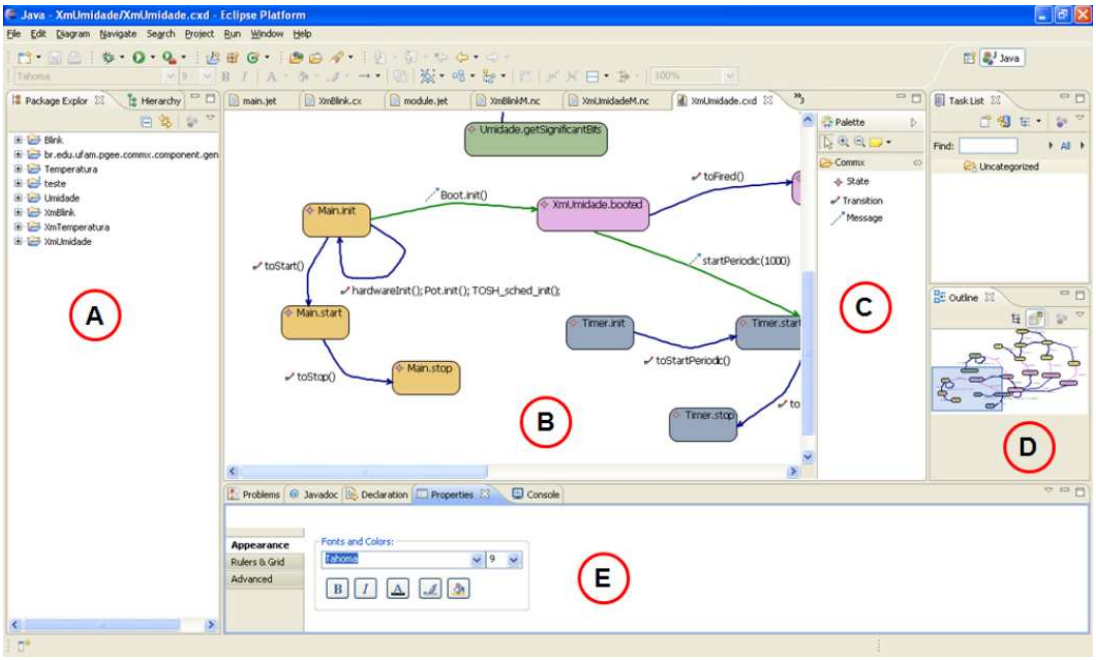


Figura 3.3: Ambiente de desenvolvimento utilizando o *X-Machine*. (Braga, 2012)

desenvolvedor alterar características dos componentes na *Modeling View* (D), e pelo **Outline View** (E), que oferece uma visão geral da distribuição dos componentes no modelo.

A geração do código para nesC ocorre por meio de duas traduções intermediárias: primeiro, a geração da representação gráfica da aplicação para a linguagem XMDL e, em seguida, para XML. Por fim, o XML é traduzido para nesC.

No *X-Machine* é utilizada uma interface gráfica para a especificação do sistema. Entretanto, o conceito de estados abordado nessa proposta está diretamente relacionado ao nível de hardware, algo que pode tornar o desenvolvimento complexo em aplicações muito extensas. Diferentemente, na plataforma proposta nesta dissertação, o SMDM-SD, o conceito de estados está relacionado ao nível de software, sendo compreendido como um bloco de código funcional.

3.4 WISELIB

O Wiselib (Baumgartner et al., 2010) é um *framework* voltado ao desenvolvimento de algoritmos genéricos para redes de sensores heterogêneos. Para isso, o Wiselib fornece ao pesquisador uma interface de implementação de algoritmos capaz de abstrair os detalhes específicos das linguagens de programação para sensores baseadas no C, como o TinyOS, o ScatterWeb, e o Contiki [1].

A arquitetura do Wiselib, ilustrada na Figura 3.4, utiliza o mecanismo de **conceitos** (implementados por meio de modelos) para criar um código independente de plataforma. Os conceitos são divididos em três: as **Facetas do SO**, responsáveis por fornecer uma interface que abstrai o código específico de cada linguagem de programação, as **Estruturas de Dados**, que implementam internamente as estruturas de dados fornecidas pelo Wiselib e geram sua correspondência na plataforma destino requerida e a **Implementação do Algoritmo** que realiza a junção dos conceitos descritos anteriormente com o código específico de cada algoritmo.

[1]Sistema Operacional Contiki: `http://www.contikios.org`

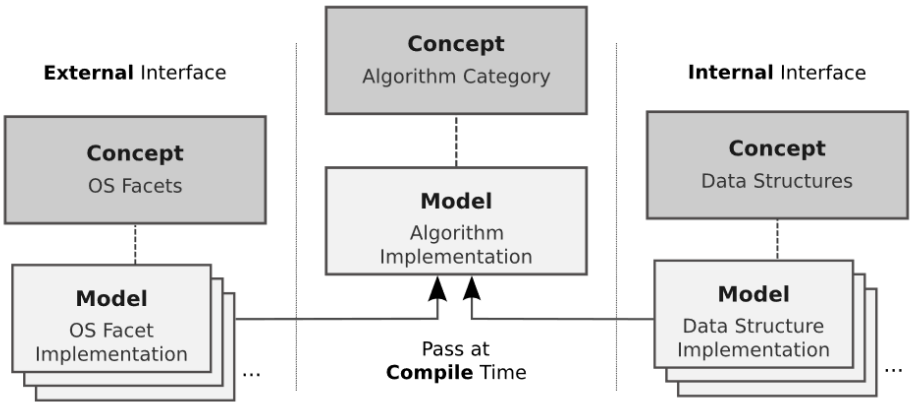


Figura 3.4: Arquitetura do Wiselib. (Baumgartner et al., 2010)

Utilizando o conceito de *templates* da linguagem de programação C++, o Wiselib é capaz de traduzir algoritmos implementados em sua plataforma para sistemas operacionais com linguagens de programação baseadas em C. Assim como o SMDM-SD, o Wiselib também utiliza o conceito de componentes para auxiliar no desenvolvimento e aproveitamento de códigos das aplicações. Porém, a ferramenta não propõe um modelo para o desenvolvimento do fluxo de execução da aplicação.

3.4.1 CBCWSN

Uma das bibliotecas do Wiselib é o CBCWSN (*Component Based Clustering in Wireless Sensor Networks*) (Amaxilatis et al., 2011). Essa biblioteca contém algoritmos para o desenvolvimento de sistemas baseado em *Clusters*. Nessa biblioteca os sistemas de armazenamento são construídos por meio da divisão da lógica dos algoritmos de agrupamento em três partes, de acordo com a funcionalidade fornecida. Cada partição é projetada para que possa progredir seu trabalho de maneira relativamente independente, garantindo a funcionalidade correta do algoritmo.

A primeira divisão é a funcionalidade ***Cluster-head Selection***, que contém algoritmos responsáveis pela eleição do líder do cluster. A funcionalidade ***Node Grouping*** possui algoritmos que realizam a união dos nós sensores ao cluster. Já a terceira partição, ***Iterador***, está relacionada à organização dos nós enquanto as decisões de cluster são tomadas por cada nó. Utilizando a arquitetura do Wiselib o desenvolvedor pode construir os sistemas de armazenamento conectando os algoritmos escolhidos em cada funcionalidade.

3.5 IOTSUITE

A plataforma IoTSuite (Soukaras et al., 2015), auxilia no desenvolvimento de aplicações para IoT propondo um método para construção de DSLs. Ou seja, o IoTSuite utiliza uma linguagem de alto nível com gramática customizada para cada domínio de aplicação. Por exemplo, uma aplicação de automação predial é fundamentada em termos como quartos e pisos. Assim, o especialista do domínio pode definir a gramática com termos que melhor representem a natureza da aplicação que será desenvolvida.

As etapas de desenvolvimento no IoTSuite, ilustradas na Figura 3.5, são: (1) especificação da gramática de acordo com o domínio, (2) compilação da especificação da gramática, (3)

especificação da arquitetura da aplicação, (4) compilação da especificação da arquitetura, (5) implementação da lógica do aplicativo, (6) especificação da implantação de destino, (7) mapeamento, (8) implementação de *drivers* de dispositivo e (9) conexão com os dispositivos

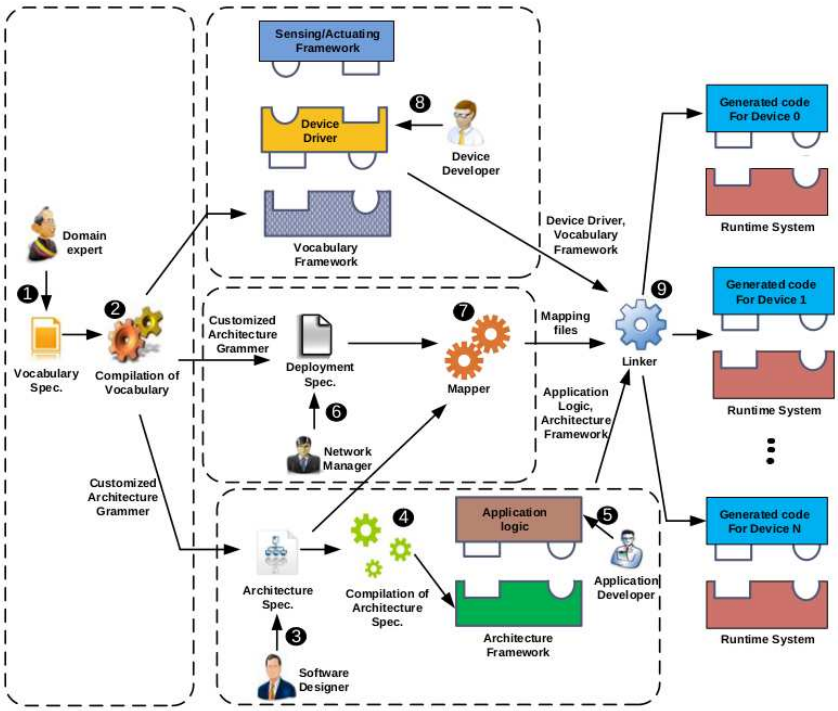


Figura 3.5: Fases de desenvolvimento utilizando o IoTSuite. (Soukaras et al., 2015)

Apesar das facilidades que o IoTSuite oferece ao desenvolvedor como a possibilidade de utilizar uma gramática próxima à funcionalidade dos dispositivos, a plataforma requer o trabalho de um especialista para configuração de uma nova linguagem para cada domínio. A versão atual do IoTSuite gera código Java e realiza a implementação em dispositivos habilitados com Android, JavaSE e o *middleware* MQTT.

3.6 MODELO BIP

No trabalho de Lekidis et al. (2018) é apresentado o **BIP** (*Behaviour, Interaction, Priority* - Comportamento, Interação, Prioridade), um modelo de desenvolvimento para a especificação do fluxo de execução de sistemas IoT. Nessa proposta também é apresentada a DSL BIP, desenvolvida para implementar a coordenação do comportamento de um conjunto de componentes atômicos (componentes que não podem conter outros componentes).

Apesar da DSL gerar códigos em C para aplicações nativas do SO Contiki, o BIP pode ser utilizado para modelar sistemas em RSSF independente da linguagem, como o exemplo introduzido em (Basu et al., 2007), que mostra o uso do BIP na modelagem de sistemas para o SO TinyOS. No sistema modelado pelo BIP, cada ação é representada como um comportamento (***B**ehaviour*). A mudança de uma ação para outra trata-se de um processo semelhante à transição de estados, chamado (***I**nteraction*). A definição da coordenação das ações é realizada por meio das interações, que por sua vez possuem seus níveis de prioridades (***P**riority*).

A Figura 3.6 ilustra a modelagem do fluxo de execução de um sistema de detecção de movimento utilizando a modelagem BIP. É possível notar que a cada transição uma nova ação é realizada. Por exemplo, quando um movimento é detectado (*motion_detected*) é realizada uma

transição para a tomada da próxima ação, que tem como objetivo verificar o horário de trabalho no ambiente monitorado (*working_hours*). Caso seja horário de trabalho, a ação subsequente é acender as luzes (*switch_on_lights*). Porém, se não for horário de trabalho, é realizada a ação de tocar alarme (*trigger_alarm*).

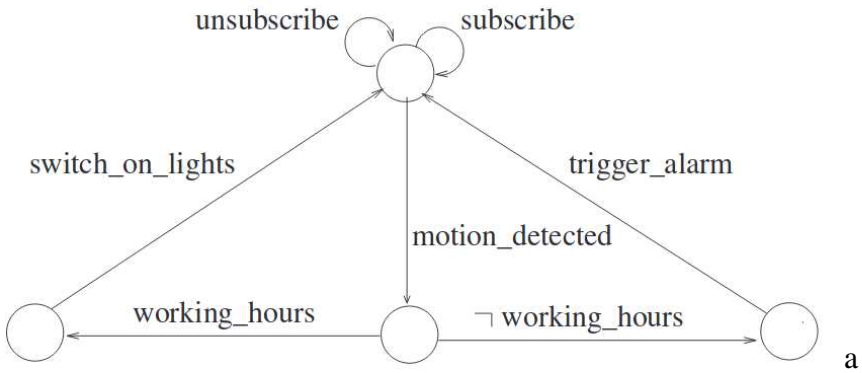


Figura 3.6: Modelagem de uma aplicação de detecção de movimentos utilizando o BIP. (Lekidis et al., 2018)

Nota-se que o processo de desenvolvimento de sistemas adotado pelo *framework* BIP é semelhante ao proposto nesta dissertação, visto que é utilizado um modelo baseado em máquina de estados para especificação do coordenador, e uma DSL para a implementação do fluxo de execução em dispositivos sensores. Entretanto, o modelo adotado pelo BIP compreende os estados da ME como ações isoladas. Já a proposta apresentada nessa dissertação propõe a componentização das entidades da aplicação e a utilização dos estados como um conjunto de ações.

3.7 RCBM

O RCBM (*Reusable Component-Based Model*) (Carrero et al., 2017) é um modelo de desenvolvimento de sistemas de armazenamento de dados para simulação. No RCBM é proposta uma estrutura que apoia a construção de componentes reutilizáveis, com o objetivo de auxiliar o desenvolvedor na reutilização de código. Essa proposta foi elaborada com base no método formal de máquina de estados com transições lógicas. Esse método é adequado para a especificação e implementação de sistemas para RSSF, pois é capaz de representar o comportamento da rede.

São propostas três categorias de componentes, como ilustrado na Figura 3.7: (i) de biblioteca, (ii) de aplicação, e (iii) de coordenação. Os componentes de biblioteca são genéricos e nativos do *framework*. Esses componentes fornecem funções úteis para o desenvolvimento de outros componentes, sejam de aplicação ou de coordenação. Por exemplo, o componente de agregação fornece funções para agregar e classificar conjuntos de dados.

Componentes de aplicação representam entidades do sistema. Ao separar a entidade do sistema do código geral da aplicação, é possível definir suas funcionalidades em uma classe específica. Fazendo isso, o desenvolvedor pode reaproveitar o componente sempre que necessário, além de facilitar a implementação de outros algoritmos que utilizem a mesma entidade. Por fim, o componente de coordenação é o responsável por orquestrar o fluxo de execução da aplicação. Para auxiliar o desenvolvedor na especificação e implementação do componente coordenador, a DSL SLEDS (Carrero et al., 2018) foi proposta. Essa linguagem foca nos sistemas de armazenamento de dados em RSSF, e gera códigos para o simulador NS-2.

Nessa dissertação a DSL SLEDS foi reimplementada para geração de códigos para RSSF reais, atendendo os requisitos da linguagem nesC. Assim, é proposta a linguagem SLEDS-SD.

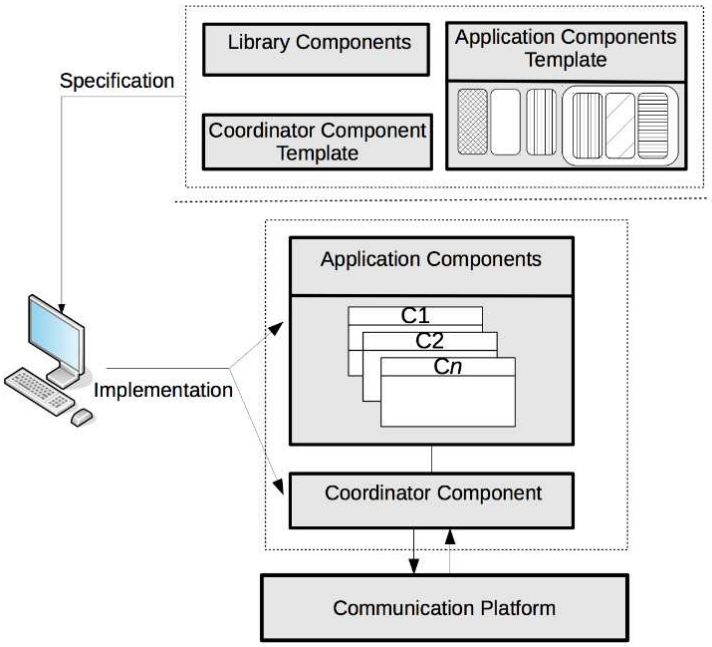


Figura 3.7: Arquitetura do RCBM. (Carrero et al., 2017)

Nessa nova linguagem é considerada a tipagem estática e a geração dos componentes de aplicação na etapa de implementação. No Capítulo 4 é apresentada essa nova arquitetura.

3.8 CONSIDERAÇÕES FINAIS

Foram descritas propostas existentes na literatura com objetivos semelhantes ao SMDM-SD e ao SLEDS-SD, que geram códigos para SOs da IoT e das RSSFs. Na Tabela 3.1 é feito um resumo das principais características dos ambientes de desenvolvimento para aplicações de RSSF apresentados neste capítulo. Os trabalhos são classificados de acordo com o escopo, o método formal adotado, o modelo de desenvolvimento, o código gerado, e o sistema operacional de sensor ao qual o sistema é implantado. Na Tabela 3.1 também é introduzido o modelo proposto, o SMDM-SD, cuja linha está pintada em cinza.

A linguagem **SenNet** (Salman e Al-Yasiri, 2016) foi proposta para auxiliar o desenvolvedor na programação e implementação de aplicações para RSSFs. Nessa proposta, foi adotado o Desenvolvimento Orientado a Modelos (MDD). Os modelos são definidos com base nas funções do sensor na topologia da rede. Para programar a aplicação, a DSL SenNet abstrai os detalhes de baixo nível do código que é gerado no nesC. Além de adotar um modelo de desenvolvimento diferente, o SenNet se diferencia do SMDM-SD por não se preocupar com o reaproveitamento de código, que é um dos objetivos do nosso trabalho.

O modelo de desenvolvimento **Tokenit** (Taherkordi et al., 2015) segue uma abordagem baseada em estado com transições tokenizadas. Nesta proposta o programador desenvolve as atividades, denominadas tokens, na linguagem C. Em seguida, o fluxo de execução é especificado em XML para descrever o modelo TOKENIT. Os tokens e o modelo são processados para gerar um programa em C, com o código da atividade em cada estado. O programador então precisa revisar o código e fazer pequenas modificações para intercalar o código relacionado com o código funcional.O Tokenit gera código C para ser implantado em dispositivos com sistema operacional Contiki. Apesar de usar a mesma ideia de separar o fluxo de execução do aplicativo do código

Tabela 3.1: Descrição dos trabalhos relacionados

Trabalho	**Escopo**	**Método Formal**	**Modelo de Desenvolvimento**	**Código Gerado**	**Sistema Operacional**
SenNet	RSSF	Meta-modelo	Baseado em modelos de rede hierárquicos	nesC	TinyOS
Tokenit	RSSF e IoT	Máquina de estados	Baseado em estados como métodos	C	Contiki
X-Machine	RSSF	Máquina de estados	Baseado em estados como abstrações de hardware	nesC e XML	TinyOS
IotSuite	IoT	Meta-modelo	Gramática adaptativa conforme o domínio	Java	Android e J2SE
Wiselib	RSSF	Meta-modelo	Baseado em templates genéricos	Derivado do C	TinyOS, Contiki e ScatterWeb
BIP	IoT	Máquina de estados	Baseado em estados como componentes	C	Contiki
RCBM	Sistemas de armazenamento para RSSF	Máquina de estados com transições lógicas	Baseado em estados como métodos e em componentes	Otcl	NS-2
SMDM-SD	Sistemas de armazenamento para RSSF	Máquina de estados com transições lógicas	Baseado em estados como métodos e em componentes	nesC	TinyOS

específico de cada token, o Tokenit requer algum esforço para descrever o modelo em XML e, em seguida, intercalar manualmente o código gerado. Em contraste, nosso trabalho tem a ideia de usar componentes para fornecer serviços e uma linguagem próxima ao modelo de especificação para programar o fluxo de execução da aplicação.

O **X-Machine** (Braga, 2012) é um modelo de desenvolvimento para aplicações em RSSFs. O modelo é baseado no formalismo Communicating X-Machine (CMX). Os estados representam componentes de hardware e estão diretamente associados às suas funcionalidades. Por exemplo, o sensor de umidade de um dispositivo pode ser modelado como um estado. Os programas X-Machine são traduzidos para nesC para serem instalados em dispositivos que executam o TinyOS. Embora o X-Machine e o modelo proposto SMDM-SD sejam ambos baseados em máquinas de estado, o primeiro visa a aplicação no nível de hardware, enquanto o SMDM-SD considera os estados no nível de software, representando blocos de código funcionais

O ambiente de desenvolvimento **IoTSuite** (Soukaras et al., 2015) é formado por um conjunto de ferramentas que auxiliam no desenvolvimento de aplicações para a IoT. O IoTSuite gera código em Java para ser instalado em dispositivos com Android, JavaSE ou MQTT. O desenvolvimento adota uma linguagem de alto nível com sintaxe customizada de acordo com o domínio do aplicativo. Essa abordagem fornece generalidade, mas requer o trabalho de um especialista para configurar uma nova linguagem para cada domínio. O SMDM-SD, por outro lado, visa o desenvolvimento de sistemas de armazenamento e adota um modelo baseado em máquina de estado, que é intuitivo e adequado para o desenvolvimento de tais sistemas.

O framework **Wiselib** (Baumgartner et al., 2010) foi proposto para o desenvolvimento de aplicações para RSSFs com um código genérico, que pode produzir códigos para diversas

plataformas, como TinyOS e ScatterWeb. A arquitetura do Wiselib adota um mecanismo de conceito, implementado por meio de modelos. Os conceitos são responsáveis por fornecer uma interface que abstrai o código de baixo nível de cada linguagem de programação. Apesar de propor um método eficiente de reutilização de código específico, ao contrário desta dissertação, o Wiselib não propõe um modelo para o desenvolvimento do fluxo de execução da aplicação.

O ambiente de desenvolvimento **BIP** (Behavior, Interaction, Priority) (Lekidis et al., 2018) consiste em um modelo de desenvolvimento de sistemas IoT que utiliza os conceitos de comportamento, interação e prioridade para projetar o fluxo de execução. O BIP também usa uma DSL baseada em máquina de estado. A ferramenta gera código para o sistema operacional Contiki. Embora o modelo de desenvolvimento do BIP seja semelhante ao proposto nesta dissertação, ele não oferece suporte ou utiliza o conceito de componentes que promovem a reutilização do código.

Em (Hussein et al., 2017), outra abordagem orientada à máquina de estado é proposta para auxiliar na modelagem e implementação de sistemas IoT adaptativos. Os estados modelam diferentes configurações do sistema, enquanto as transições representam seus gatilhos de adaptação, como falhas de hardware. Com base no modelo de projeto do sistema, o código para a plataforma MicroEJ é gerado. Apesar de adotar o mesmo modelo de desenvolvimento, o objetivo deste trabalho é a adaptabilidade dos sistemas, enquanto do SMDM-SD é o desenvolvimento de modelos de armazenamento.

O SMDM-SD compartilha com o **RCBM** (Carrero et al., 2019) os mesmos princípios de reutilização de código e uma DSL baseada em máquinas de estado. Na verdade, SLEDS-SD é uma adaptação da linguagem SLEDS proposta para se ajustar às restrições de linguagens para dispositivos sensores, como o nesC. Isso ocorre porque o SLEDS foi projetado para gerar o código de simulação NS-2, que é baseado em C++ e, portanto, oferece suporte a tipos dinâmicos . Este recurso facilita o desenvolvimento de software baseado em componentes, mas geralmente não existe em linguagens para sensores de programação. Como resultado, o SLEDS-SD modifica todas as construções na linguagem que depende de tipagem dinâmica. Além disso, o SLEDS-SD promove um desenvolvimento de cima para baixo dos componentes da aplicação porque o processo de tradução de SLEDS-SD para nesC também gera um conjunto de componentes de interface que podem posteriormente ser implementados pelo programador. É planejado no futuro implementar uma tradução do SLEDS-SD para o NS-3. Fazendo isso, o mesmo código pode ser usado tanto para avaliação por simulação quanto para implantação em uma RSSF real.

4 O MODELO DE DESENVOLVIMENTO SMDM-SD

Neste capítulo é apresentado o SMDM-SD, o modelo de desenvolvimento proposto nessa dissertação. Esse modelo é baseado em componentes reutilizáveis desenvolvidos a partir da entidade das aplicações. O SMDM-SD é adequado para o desenvolvimento de sistemas de armazenamento para RSSFs, pois utiliza uma máquina de estados para especificação da aplicação e um método formal que representa bem a natureza de aplicações reativas, além de promover o reuso de código com a componentização do sistema.

4.1 ETAPAS DE DESENVOLVIMENTO

O modelo de desenvolvimento SMDM-SD é ilustrado na Figura 4.1. Esse modelo é composto de 3 etapas: (1) especificação usando uma Máquina de Estados, (2) desenvolvimento de um programa de orquestração em SLEDS-SD e componentes adicionais, e (3) geração de código na linguagem do sensor de destino e implantação do sistema.

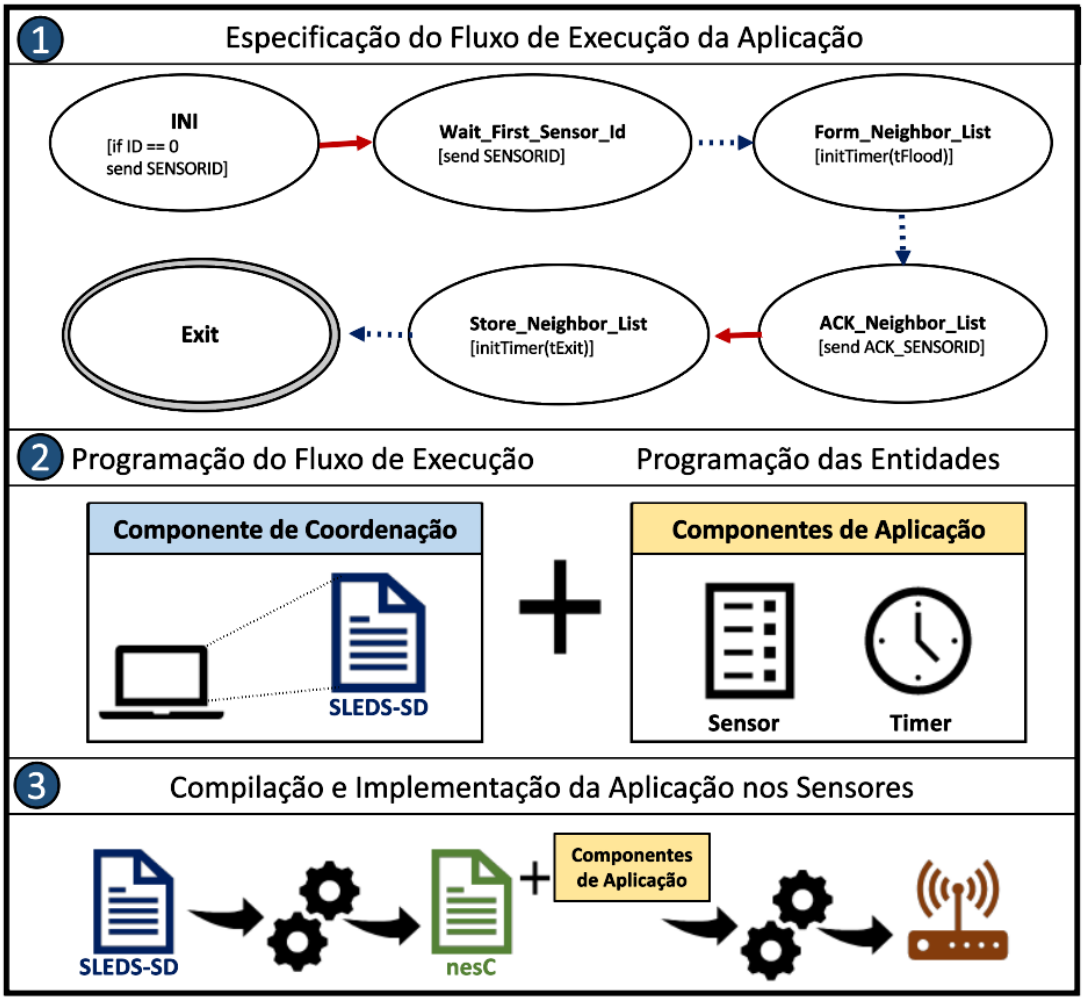


Figura 4.1: Modelo de Desenvolvimento SMDM-SD

Nas próximas seções (4.2, 4.3 e 4.4) cada etapa é detalhada, juntamente com a DSL SLEDS-SD.

4.2 ETAPA DE ESPECIFICAÇÃO

Na especificação de sistemas para RSSFs, um modelo amplamente utilizado é sua representação como uma máquina de estados (Ouadjaout et al., 2014), dada a natureza reativa dos dispositivos sensores. No SMDM-SD, as mudanças de estado são disparadas não apenas por eventos, mas também podem modelar o fluxo de execução da aplicação por meio de transições *lógicas*. Considerando a máquina de estados com transições lógicas e baseadas em eventos, é possível criar uma especificação de alto nível de um programa, na qual cada estado define uma funcionalidade simples, que corresponde aproximadamente a uma função (ou procedimento) em linguagens de programação tradicionais.

Foram considerados apenas dois tipos de transições baseadas em eventos: disparada por uma solicitação, como o recebimento de uma mensagem, e disparada por um temporizador. Também foi pressuposto que todos os sensores em uma RSSF estão executando a mesma aplicação e que eles se comunicam via rádio, por meio de transmissões de mensagens multi-hop. Assim, sempre que um sensor envia uma mensagem, todos os vizinhos dentro de sua faixa de transmissão de rádio a recebem. Porém, é importante notar que em cada dispositivo sensor individualmente, não há relação entre o estado em que uma mensagem é enviada e o estado em que a recebe. Ou seja, se um sensor envia uma mensagem do estado s, nem sempre seus vizinhos estão no mesmo estado s. Na verdade, cada um deles pode estar em estados diferentes e tratar a mensagem de maneiras diferentes. Como resultado, na máquina de estados, os estados têm um nome e podem ser associados a dois tipos de ações: enviar uma mensagem ou iniciar um cronômetro. Receber uma mensagem (ou um tempo limite do cronômetro) aciona uma transição baseada em evento que não necessariamente começa a partir do estado que enviou a mensagem (ou que iniciou o cronômetro). As transições baseadas em eventos são rotuladas com seu evento de acionamento (tipo de mensagem de cronômetro), enquanto as transições lógicas são opcionalmente rotuladas com uma expressão lógica. Um exemplo é mostrado na Figura 4.2.

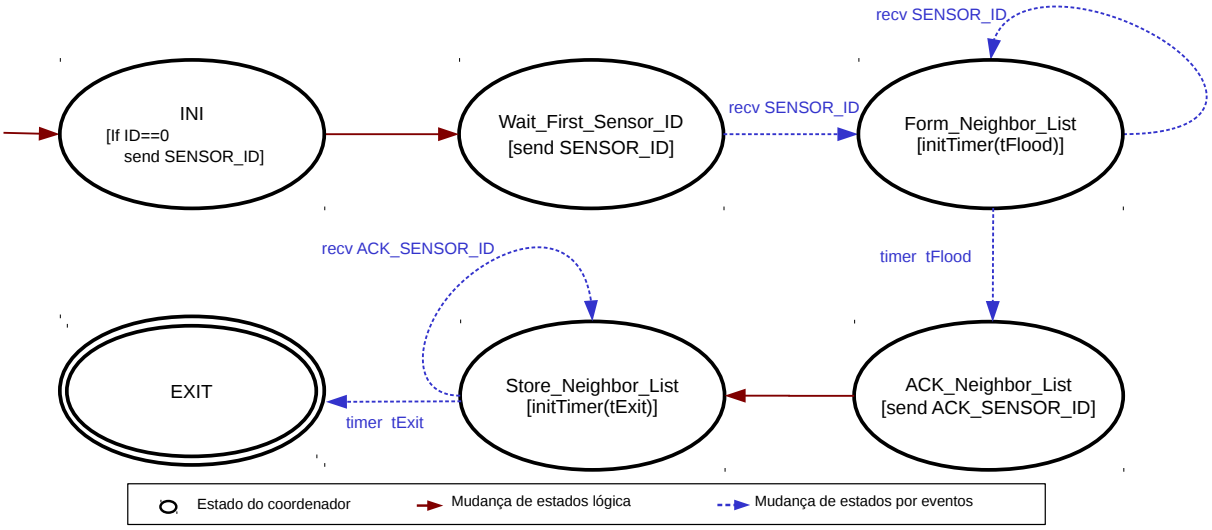


Figura 4.2: Um modelo de máquina de estados para descoberta de vizinhos por inundação. (Carrero et al., 2017).

A máquina de estado obtem, para cada sensor, seu conjunto de vizinhos. Ela começa com o estado *INI*, no qual o sensor com ID 0 envia uma mensagem `SENSOR_ID` em broadcast. Todos os sensores fazem uma transição lógica (representada em vermelho e em linha contínua) para o estado *Wait_First_Sensor_ID*. Nesse estado, o recebimento da primeira mensagem `SENSOR_ID` aciona o sensor para transmitir sua própria mensagem `SENSOR_ID` e fazer uma transição baseada

em evento (representada em azul e em linha tracejada) para o estado *Form_Neighbor_List*. Neste estado, um novo temporizador `tFlood` é iniciado. Durante este tempo ele continua recebendo mensagens `SENSOR_ID` para criar sua lista de vizinhos. Após o tempo limite de `tFlood`, uma transição de evento é feita para o estado *ACK_Neighbor_List*. Neste estado, os sensores enviam uma mensagem de confirmação aos seus vizinhos e fazem a transição lógica para o estado *Store_Neighbor_List*. Nesse estado, os sensores recebem mensagens `ACK_SENSOR_ID` de seus vizinhos e armazenam seus IDs durante o tempo `tExit`. Após o tempo limite, a aplicação faz uma mudança para o estado *EXIT*.

4.3 ETAPA DE PROGRAMAÇÃO

Com base na especificação da máquina de estados, um programa correspondente é desenvolvido na linguagem específica de domínio SLEDS-SD. Um programa SLEDS-SD consiste em um conjunto de estados, que por sua vez, consiste em uma sequência de ações. Um exemplo de implementação de estado pode ser encontrado na Listagem 4.1. Ela corresponde ao estado *Form_Neighbor_List* na Figura 4.2. Observe que, o comando *during (timer) on recvBroadcast* na linguagem SLEDS-SD indica a inicialização de um *timer* e o recebimento de mensagens durante este período de tempo. No exemplo, durante o tempo *tFlood*, a aplicação recebe mensagens do tipo *SENSOR_ID*. A cada novo recebimento, o ID do sensor recebido na mensagem é armazenado em uma lista (*listSensorAnnouncements*). Finalmente, quando o temporizador expira, uma transição é feita para o estado *Ack_Neighbor_List*. Vale notar a similaridade entre a especificação da máquina de estados e as construções da linguagem, que serão detalhadas no Capítulo 5.

```
1 STATE Form_Neighbor_List() {                                          //Declaração de um estado
2     during (tFlood) on recvBroadcast(SENSORID, msdID, pktSensorId){   //Action
3         listSensorAnnouncements. insert (pktSensorId->id);}           //Inserindo valor na lista
4     nextState ACK_Neighbor_List()}                                    // Mudança de estado
```

Listagem 4.1: Implementação do Estado Form_Neighbor_List.

Ao programar os estados, algumas funções fornecidas por componentes existentes ou componentes a serem implementados podem ser chamadas. Existem dois tipos de componentes: componentes de biblioteca e componentes de aplicação. Os componentes da biblioteca fornecem funções genéricas e úteis para o desenvolvimento das aplicações. Algumas linguagens, como nesC, já fornecem alguns, como o componente *timer*, que pode ser usado para gerar eventos em intervalos regulares de tempo. Componentes adicionais de biblioteca podem ser implementados, como o componente de agregação de dados, para fornecer funções para calcular um valor de agregação dos elementos de uma lista, como soma, média, máximo e mínimo. Os componentes de aplicação estão diretamente associados às entidades no domínio do aplicação. Uma vez que o SMDM-SD foca em modelos de armazenamento de repositório, os componentes de aplicação podem incluir: (i) Sensor, (ii) Gateway e (iii) Cluster. Por exemplo, o Componente **Cluster** pode ter funções para escolher *Cluster-Heads* (CHs) e construir *Clusters* (Akyildiz et al., 2002).

Em SLEDS-SD, programas que usam componentes devem conter instruções `use` explícitas no início do programa. Os componentes de biblioteca estão prontos para serem reutilizados por qualquer aplicação. Os componentes de aplicação, por outro lado, podem não ter sido implementados anteriormente. Para apoiar a ideia de reutilizar componentes existentes, e também permitir a definição de novos componentes, o SMDM-SD usa a seguinte estratégia. Se o componente existir, os tipos são verificados, da mesma forma que os compiladores tradicionais. Porém, caso não existam, um arquivo de cabeçalho com suas interfaces é gerado para orientar o desenvolvedor sobre como preencher a aplicação. Essa estratégia, ao mesmo tempo em que promove a reutilização do código, ajuda o desenvolvedor a determinar quais os componentes de

aplicação precisam ser desenvolvidos. Além disso, esta estratégia é apropriada para a geração de código em uma linguagem com tipos estáticos, utilizada pela maioria das linguagens para sensores.

O componente *Cluster* é formado por diversas funcionalidades, como a seleção do CH e o agrupamento sensores em clusters. A Listagem 4.2 apresenta um exemplo da interface para a entidade *Cluster*. Foram definidas duas funções: a eleição do CH (`selectCH`) e a associação de um membro a um *cluster* (`join`).

```
1 interface Cluster {
2     command int selectCH( int neighbors [] );
3     command int join( int candidates [] ); }
```

Listagem 4.2: Template do componente *Cluster*.

A implementação dos componentes de aplicação é dependente do modelo de armazenamento que o desenvolvedor está implementando. Dessa maneira, com o uso do SMDM-SD, é possível separar o código das funcionalidades das entidades do restante da aplicação. No estudo de caso apresentado no Capítulo 6 são apresentados os modelos de agrupamento LCA e LEACH. As diferenças fundamentais entre esses modelos são os métodos de seleção de CH e de associação de sensores aos *clusters*. Assim, ao realizar a implementação, o código que distingue os modelos se concentra nas funcionalidades, possibilitando que boa parte do código de coordenação possa ser aproveitado.

Os componentes de aplicação estão na camada de implementação, pois necessitam do arquivo de *interface* gerado pelo coordenador para serem desenvolvidos. De certa forma o processo de desenvolvimento do coordenador e dos componentes de aplicação ocorre de maneira paralela, pois no processo de desenvolvimento do coordenador, o desenvolvedor pode identificar novos componentes de aplicação, bem como novas funções.

4.4 ETAPA DE COMPILAÇÃO E IMPLANTAÇÃO DA APLICAÇÃO NOS SENSORES

Na última etapa do modelo de desenvolvimento, o código SLEDS-SD é compilado e traduzido para uma linguagem de programação de sensor de destino. Na implementação atual, é gerado código para a plataforma TinyOS. Assim, o compilador SLEDS-SD gera o código nesC que corresponde à máquina de estado especificada, junto com um arquivo de cabeçalho com os componentes da aplicação que devem ser desenvolvidos para completar a aplicação no nesC. O conjunto de arquivos é então fornecido como entrada para o compilador nesC, para gerar o arquivo executável da aplicação a ser implantado nos dispositivos sensores. Os detalhes da geração de código são descritos na seção 5.2 do capítulo 5.

O compilador de códigos SLEDS-SD, quando utilizado para tradução em nesC, também gera a interface dos componentes de aplicação que não foram desenvolvidos. Portanto, o desenvolvedor deve implementar os componentes de aplicação antes da próxima etapa, a compilação do código nesC e dos componentes utilizando o compilador do TinyOS para implantação da aplicação nos dispositivos de RSSFs em ambientes reais.

4.5 CONSIDERAÇÕES FINAIS

O SMDM-SD simplifica o desenvolvimento de novos sistemas de armazenamento para RSSFs através de uma separação clara entre a especificação, implementação e implantação. Cada etapa do desenvolvimento possui estratégias especificas para execução ágil.

Na **especificação** é adotado o método formal máquina de estados com transições lógicas e por eventos. Esse formalismo é utilizado para representar o fluxo de execução de sistemas de armazenamento em RSSF. Já na **programação** é proposta uma linguagem de programação de alto nível que se assemelha muito ao modelo de especificação, o SLEDS-SD. Além disso, com a criação de componentes conectáveis, nessa etapa também é possível reaproveitar códigos já desenvolvidos. Na ultima etapa, **implantação da aplicação**, a geração de códigos do coordenador e dos componentes de aplicação através do compilador de SLEDS-SD para nesC, ajuda o desenvolvedor a preparar os arquivos para a implantação da aplicação em dispositivos que utilizam o TinyOS. A seguir, no Capítulo 5, são apresentadas as construções da linguagem SLEDS-SD, bem como a geração de códigos em nesC.

5 A LINGUAGEM SLEDS-SD

Este capítulo apresenta a linguagem SLEDS-SD e sua tradução para o nesC. A maioria de suas construções deriva do SLEDS (Carrero et al., 2019), uma DSL proposta para gerar código de *simulação* para o NS-2. Assim, este capítulo foca nas principais construções da linguagem e as adaptações feitas para permitir sua tradução para o nesC.

5.1 COMPONENTES DA LINGUAGEM

Um programa SLEDS-SD consiste em três partes: (i) declarações de importação de componentes; (ii) declarações de dados; e (iii) definições de estados, conforme ilustrado na Listagem 5.1. Dentre as declarações de dados, é necessário que a estrutura das mensagens transmitidas entre os sensores, bem como seus identificadores sejam definidos utilizando a sintaxe mostrada abaixo:

messageType::= **MESSAGE TYPE** '**{**' identifier ('**,**' identifier)* '**}**' '**;**'
message::= **MESSAGE** identifier '**{**' typeList '**}**' '**;**'

```
1  //Declaração das bibliotecas importadas (use .. as ..)
2  use compSensor as ComponentSensor;
3  use compLibMSG as ComponentLibMessage;
4
5  //Declaração dos identificadores das mensagens (message type)
6  message type { SENSORID, MAXWINNER, MINWINNER };
7
8  //Declarações de estruturas de mensagens (message)
9  message msgSensorId { int id; };
10 message msgMaxWinner { int maxWinnerId; };
11 message msgMinWinner { int minWinnerId; };
12
13 program MaxMinCoordinator ( ) {          //Declaração do programa (program)
14     const tCluster =25;                  //Declaração de constante (const)
15     msgSensorId pktSensorId;             //Declaração de uma mensagem do tipo (msgSensorId)
16     int round, myID, winnerID, myCH;     //Declaração de inteiros (int)
17     list <int> listSensorAnnounce [10], sensorList [5]; //Declaração de listas de inteiros (list <int>)
18
19     STATE ini( ){                        //Declaração de um estado (STATE)
20         winnerID=compSensor->getSensorId();
21         nextState FloodMax(winnerID);
22     }}
```

Listagem 5.1: Estrutura de um programa SLEDS-SD.

A Listagem 5.1 fornece exemplos de declarações de estruturas de mensagens (linhas 9-11), de identificadores de mensagens (linha 6) e de uma variável do tipo `msgSensorId` (linha 15). A declaração **message type** é semelhante a um tipo enumerado em C e define um conjunto de identificadores de tipo de mensagem. Eles auxiliam na implementação de eventos de recebimento de mensagens, permitindo que diferentes tipos de mensagens sejam tratados adequadamente. As declarações **message** definem novos tipos na linguagem, com as estruturas das mensagens. As primitivas da linguagem para troca de mensagens (*send*, *broadcast*, *recv* e *recvBroadcast*) possuem um parâmetro deste tipo.

Todos os programas devem ter um estado `ini`, no qual a execução começa. As primitivas de linguagem para implementar estados incluem as instruções de fluxo de controle tradicionais, como *if*, *for* e *while*, bem como aquelas relativas a eventos e transições de estado:

- **nextState**: ação que faz uma transição para outro estado;
- **broadcast**: ação para transmitir uma mensagem de broadcast;
- **send**: ação para transmitir uma mensagem a um sensor específico;
- **on recvBroadcast**: ação para receber uma mensagem de transmissão;
- **on recv**: ação para receber uma mensagem específica;
- **during**: ação para acionar um cronômetro.

A linguagem também oferece suporte às ações compostas, que foram identificadas como comumente usadas para desenvolver modelos de dados para RSSFs. Elas incluem o recebimento de mensagens durante um período de tempo:

- **during** (*time*) **on recvBroadcast** (*message*) **{***ActionList***}** **nextState** *state*: ação composta para receber mensagens broadcast por um tempo especificado, seguido por uma transição de estado.
- **during** (*time*) **on recv** (*message*) **{***ActionList***}** **nextState** *state*: ação composta para receber mensagens específicas por um determinado período de tempo, seguido por uma transição de estado.

Um estado com uma ação composta é mostrado na Listagem 4.1. Neste exemplo, é possível notar que os parâmetros para a *action* `recvBroadcast` são: o tipo de mensagem, uma identificação de mensagem única e uma variável de mensagem, conforme declarado na Listagem 5.1. As instruções `recv, broadcast` e `send` têm os mesmos parâmetros. Além disso, a variável `listSensorAnnounce` foi declarada na Listagem 5.1 como uma lista de elementos do tipo `int`, que tem `insert` entre suas funções associadas. Definições de listas de todos os tipos atômicos na linguagem, bem como tipos de mensagens, estão entre as extensões necessárias na linguagem SLEDS-SD para permitir a tradução para nesC. A sintaxe da linguagem SLEDS-SD encontra-se no Apêndice A, enquanto que no Apêndice B é ilustrado um exemplo de um programa completo usando essa linguagem.

5.2 TRADUÇÃO PARA NESC

A tradução do SLEDS-SD para nesC gera um método para cada estado no código-fonte. Além disso, quando o estado envolve eventos (temporizador e recebimento de mensagem), parte do código de estado deve ser gerado em diferentes partes do programa. Isso ocorre porque o nesC requer que todas as mensagens sejam tratadas por um único evento `receive`. Além disso, um tempo limite do temporizador dispara a execução de um evento associado `fired`. Assim, a tradução de SLEDS-SD para nesC requer espalhar o código dentro dos estados em diferentes partes do programa nesC. Para exemplificar o processo, considere a máquina de estados que especifica parte do algoritmo de agrupamento do modelo de armazenamento MAX-MIN (Amis et al., 2000). Este modelo adota um fluxo de execução bastante distinto, com múltiplas rodadas de troca de mensagens para coletar informações de sensores vizinhos. Na sequência, com base

nessas informações, criam-se os clusters, e são selecionados os cluster-heads e gateways. Mais especificamente, o algoritmo MAX-MIN tem quatro etapas lógicas: *(i)* FloodMax - propagação dos maiores IDs de nós, *(ii)* FloodMin - propagação dos menores IDs de nós, *(iii)* agrupamento, e *(iv)* formação de *backbone*. A Figura 5.1 ilustra a máquina de estado para as etapas *FloodMax* e *FloodMin*, enquanto o código SLEDS-SD para a etapa *FloodMax* é mostrado na Listagem 5.2.

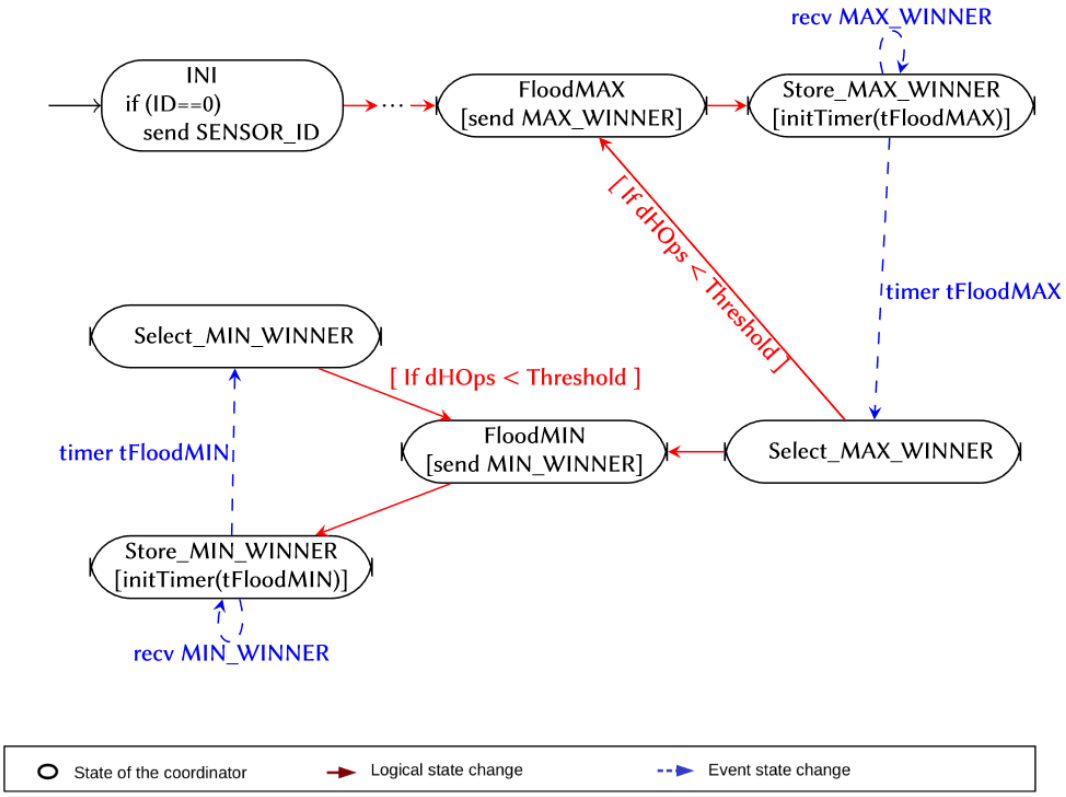


Figura 5.1: Máquina de Estados para as etapas *FloodMax* e *FloodMin* da heurística Max-Min.

Na Listagem 5.2, o parâmetro de estado *maxWinnerID* contém o identificador do sensor com o maior ID entre os recebidos nas mensagens *MAXWINNER*. A primeira ação neste estado é uma atribuição, na qual a variável *msgId* recebe um novo identificador de mensagem pela função *getNextMsgId* do componente da biblioteca de mensagens, *compLibMSG*. Em seguida, são atribuídos os valores dos elementos do pacote *pktMaxWinner*, enviado pelo comando *broadcast* da linha 4. O pacote é enviado a todos os sensores na faixa de transmissão contendo o identificador do tipo de mensagem, o identificador da mensagem, e o identificador do sensor com maior ID (*maxWinnerID*).

```
1 STATE FloodMax(int maxWinnerID) {                  //Declaração de um estado
2     msgId = compLibMSG->GetNextMsgId();            //Assigment
3     pktMaxWinner->winnerID = maxWinnerID;          // Inserir valor no pacote
4     broadcast (MAXWINNER, msgId, pktMaxWinner);    //Enviar mensagem broadcast
5     nextState StoreMaxWinner(); }                  //Mudança de estados
```

Listagem 5.2: Implementação do estado FloodMax em SLEDS-SD.

A tradução da Listagem 5.2 para nesC pode ser vista na Listagem 5.3. Este é um exemplo de tradução simples. O código nesC resultante é bastante semelhante ao original. A única característica distinta é que no nesC as declarações de pacote devem estar no início do método. Portanto, o tradutor do SLEDS-SD inclui a declaração no código resultante.

```
1  void state_FloodMax(int maxWinnerID){
2      msgMaxWinner* pktMaxWinner=(msgMaxWinner*)(call Packet.getPayload(pkt,sizeof(msgMaxWinner)));
3      currentState = FLOODMAX;
4      msgId = call compLibMSG.GetNextMsgId( );
5      pktMaxWinner->winnerID = maxWinnerID;
6      broadcast (MAXWINNER, msgId, pktMaxWinner);
7      state_StoreMaxWinner( );  }
```

Listagem 5.3: Tradução do estado FloodMax em SLEDS-SD para nesC.

A tradução não é tão simples quando eventos como recebimento de mensagens ou temporizadores estão envolvidos. Isso ocorre porque o término de um cronômetro e o recebimento de mensagens são eventos assíncronos em relação ao fluxo de execução da aplicação ou ao estado atual. Então, para realizar as ações correspondentes a esses eventos, foi introduzida a variável *currentState* no código nesC para armazenar o estado de processamento do sensor. Como exemplo, considere o estado *StoreMaxWinner*, mostrado na Listagem 5.4, que contém uma instrução composta `during-on recv`. Sua tradução para nesC é ilustrada na Listagem 5.5.

```
1  STATE StoreMaxWinner ( ) {                                          \\ Declaração de um estado
2      during (tFloodMAX) on recv(MAXWINNER, msgID, pktMaxWinner) {    \\ Action During on recv
3          ListMaxWinner.insert(pktMaxWinner->winnerID);               \\ Inserir valor na lista
4      }
5      nextState SelectMaxWinner( );                                   \\ Mudança de estados
6  }
```

Listagem 5.4: Implementação do estado StoreMaxWinner em SLEDS-SD.

Observe que o código é gerado em 3 partes diferentes. O primeiro bloco de código é gerado no método *state_StoreMaxWinner* correspondente. Neste método, o estado atual é armazenado na variável *currentState*, atribuindo-se o valor da constante *STOREMAXWINNER*. Além disso, ele inicia um cronômetro, que corresponde à instrução *during*. O segundo bloco de código é gerado no evento **receive**. Este evento gerencia todas as mensagens recebidas pelo sensor. Assim, este evento é composto principalmente de uma sequência de instruções *if-else* que verifica o valor da variável *currentState*. Esta parte do programa inclui as ações associadas à instruçõe `on Recv`. Da mesma forma, o terceiro bloco de código é gerado no evento **timer fired**, com uma estrutura semelhante. Ou seja, verifica se a variável *currentState* contém o valor *STOREMAXWINNER* e a ação associada é fazer uma mudança de estado, chamando o método *state_SelectMaxWinner*.

```
1  void state_StoreMaxWinner(){
2      currentState = STOREMAXWINNER;
3      call Timer.startOneShot(tFloodMAX);
4  }
5
6  event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len){
7      [...] // suppressed code
8      }else if( currentState == STOREMAXWINNER){
9          msgMaxWinner*pktMaxWinner=(msgMaxWinner)payload;
10         call compList.insert (ListMaxWinner, pktMaxWinner->winnerID);
11     } [...] // suppressed code
12     return msg;
13 }
14
15 event void Timer.fired (){
16     [...] // suppressed code
17     }else if( currentState == STOREMAXWINNER){
```

```
18              state_SelectMaxWinner( );
19          } [...] // suppressed code
20  }
```

Listagem 5.5: Tradução do estado StoreMinWinner em SLEDS-SD para nesC.

5.3 CONSIDERAÇÕES FINAIS

Neste capítulo foram descritas as principais construções da linguagem SLEDS-SD, bem como sua tradução para nesC. Dentre as principais construções da linguagem, as estruturas de mensagens são componentes importantes, pois elas definem a conteúdo dos pacotes de mensagens, os tipos de mensagens, e as declarações. Além disso, também foram apresentadas as declarações de estados e as principais ações da linguagem.

A tradução do código em SLEDS-SD para nesC ocorre em tempo de processamento. Para cada construção em SLEDS-SD um código correspondente em nesC é gerado. Entretanto, como em nesC a programação é baseada em eventos, o código gerado pode estar em três partes distintas, sendo dentro do próprio estado, no evento *timer* ou no *receive*. Espalhar o código em diferentes métodos e eventos torna a implementação no nesC uma tarefa difícil. Um programa em SLEDS-SD, por outro lado, apoia o desenvolvimento do programa como uma máquina de estados, permitindo que toda a lógica de cada etapa seja codificada em um único estado. Como resultado, **programas SLEDS-SD são mais compactos e fáceis de desenvolver e manter**.

A seguir, no capítulo 6, são descritos os principais resultados das avaliações de aproveitamento de código e de desempenho de aplicações desenvolvidas utilizando o modelo de desenvolvimento proposto nesta dissertação.

6 ESTUDO EXPERIMENTAL

Neste capítulo são apresentados os experimentos para validação da proposta desta dissertação. Foram conduzidos dois experimentos. O primeiro tem como objetivo determinar a eficiência do modelo de desenvolvimento SMDM-SD, com a análise do impacto da reutilização de código na implementação dos sistemas. O segundo visa determinar a eficácia da tradução, comparando os resultados da implementação com aqueles relatados em (Amis et al., 2000), utilizando os mesmos cenários e parâmetros.

6.1 IMPLEMENTAÇÃO DO SISTEMA E MODELOS DE ARMAZENAMENTO

Foi implementado um tradutor de SLEDS-SD para nesC usando Flex e Bison para desenvolver os analisadores lexico e sintático, respectivamente. O tradutor, assim como o código das aplicações apresentados neste trabalho, estão disponíveis no repositório SLEDS-SD no GitHub [1] .

Para avaliar a abordagem proposta, foi utilizado o SMDM-SD na implementação de três modelos de armazenamento: LCA, LEACH e MAX-MIN. Os algoritmos de agrupamento LCA e MAX-MIN seguem um critério de agrupamento baseado em atributos, enquanto o LEACH segue um modelo probabilístico. Os três modelos de armazenamento são baseados em repositório, que elegem alguns sensores na rede, chamados *cluster-heads*, para armazenar as leituras de um grupo de sensores (*cluster-members*), que compõem um *cluster*. Na Tabela 6.1 são mostradas as principais características de cada modelo.

Tabela 6.1: Principais características dos Modelos de Armazenamento

Modelo de Armazenamento	**Eleição de Cluster-Head**	**Formação de Cluster**
LCA	O menor ID	Cluster-Head com maior intensidade no sinal de radio
LEACH	Probabilístico dividido em rounds	Cluster-Head com maior intensidade no sinal de radio
MAX-MIN	O menor ID na lista dos maiores IDs	O primeiro CH na rota do Gateway

O LCA (Linked Cluster Algorithm) (Baker e Ephremides, 1981) é um modelo de armazenamento hierárquico destinado a ser usado para pequenas redes. O critério de agrupamento é baseado no identificador único (ID) associado a cada nó sensor. Durante a fase de eleição de CH, o LCA elege como CH um nó com o ID mais baixo entre seus vizinhos que não receberam um anúncio de CH. Após a fase de eleição, os sensores restantes se juntam ao cluster do CH mais próximo.

O LEACH (Low-Energy Adaptive Clustering Hierarchy) (Heinzelman et al., 2000) é um modelo probabilístico que forma clusters de um salto. O LEACH assume que todos os nós estão dentro do alcance de comunicação um do outro. Os sensores se elegem como Cluster-Heads com uma probabilidade p. Os sensores restantes se juntam ao cluster do CH que requer o menor consumo de energia para se comunicar.

Ao contrário dos modelos de armazenamento LCA (Baker e Ephremides, 1981) e LEACH (Heinzelman et al., 2000), que seguem um fluxo de execução de clustering de um salto

[1]SLEDS-SD Language Repository: `https://github.com/sleds-sd`

tradicional, o MAX-MIN envolve um fluxo de eventos muito distinto. Embora o fluxo de execução do algoritmo MAX-MIN seja diferente de LCA e LEACH, algumas etapas se assemelham com as desses algoritmos. A separação do coordenador dos componentes da aplicação propostos pelo SMDM-SD permitiu explorar essa similaridade para a reutilização de código, reduzindo a complexidade para projetar e implementar novos sistemas. A seguir são apresentados os resultados da avaliação de reusabilidade e da avaliação de comportamento.

6.2 ANÁLISE DE REUSABILIDADE

Esta seção descreve uma avaliação empírica que determina a quantidade de reutilização de código promovida pelo SMDM-SD. Foram analisadas as implementações dos modelos de armazenamento LCA, LEACH e MAX-MIN. Na Tabela 6.2 são descritos os resultados obtidos na avaliação da reutilização do código. É importante observar que no SMDM-SD o fluxo de execução é implementado no SLEDS-SD, enquanto os componentes são implementados no nesC.

Tabela 6.2: Reutilização de código ao implementar modelos de armazenamento

	Componentes em nesC		Coordenador em SLEDS-SD	
Modelo de Armazenamento	**Número de Linhas**	**% Linhas Reutilizadas**	**Número de Linhas**	**% Linhas Reutilizadas**
LCA	113	78.76 %	92	97.82 %
LEACH	119	74.78 %	98	95.97 %
MAX-MIN	242	32.23%	140	24.28 %

Como os modelos LCA e LEACH utilizam o mesmo fluxo de execução para a formação dos *clusters*, seus coordenadores são quase idênticos, com uma similaridade de mais de 95% das linhas de código. O código difere apenas pelos parâmetros das chamadas de funções dos componentes de aplicação, que levam em consideração os critérios distintos para a seleção de CH. Já o Max-Min possui um número maior de estados, conforme apresentado na Tabela 6.3, portanto a taxa de reutilização de código foi próxima a 25%. Na implementação dos componentes, a similaridade é em torno de 75%, uma vez que os modelos aplicam critérios diferentes para a seleção de CHs e a criação de *clusters*. Já o modelo de armazenamento MAX-MIN adota uma sequência estendida de etapas lógicas, resultando em 32% de reutilização de código dos componentes em nesC.

Na Tabela 6.3 são listados, para cada modelo de armazenamento, o número total de estados, número de linhas de código em SLEDS-SD e número de linhas de código geradas pela tradução, em nesC. A modelagem do coordenador das aplicações LCA e LEACH é composta por 7 estados. Eles representam etapas de processamento lógico, como *selectCH*, *clusterFormation* e *joinCluster*. O MAX-MIN, que possui um fluxo de execução mais complexo, é composto por 14 estados.

Tabela 6.3: Número de estados e linhas de código do coordenador por modelo de armazenamento

Modelo de Armazenamento	**Número de Estados**	**Número de linhas em SLEDS**	**Número de linhas em nesC**
LCA	7	92	153
LEACH	7	98	164
MAX-MIN	14	140	223

No desenvolvimento da aplicação LCA foram implementadas 92 linhas lógicas de código em SLEDS-SD. O código gerado no nesC possui 153 linhas. No LEACH, 98 linhas foram usadas em SLEDS-SD e foram geradas 164 em nesC. Finalmente, no Max-Min a implementação do coordenador possui 140 linhas de código em SLEDS-SD. Ao traduzir o código Max-Min em SLEDS-SD para nesC, 223 linhas lógicas de código foram geradas. Esses resultados mostram que o número de linhas de um programa SLEDS-SD é aproximadamente 40% menor que o número de linhas geradas no nesC. Isso sugere que o esforço de desenvolvimento em SLEDS-SD também é menor do que para nesC.

A separação do coordenador dos componentes da aplicação propostos pelo SMDM-SD e o desenvolvimento dos sistemas de armazenamento na linguagem SLEDS-SD mostram que a proposta potencialmente auxilia o desenvolvimento de novos modelos de armazenamento. Em modelos com fluxo de execução semelhante, como LCA e LEACH, foi obtido um alto percentual de linhas de código reutilizadas. Estes resultados seguem do fato de que o SMDM-SD fornece um modelo de desenvolvimento baseado em componentes *flexíveis* com separação do fluxo de de execução. Além disso, os mesmos componentes de aplicação podem ser reutilizados em diferentes fluxos de execução, fornecendo uma nova implementação, como para o MAX-MIN. A linguagem SLEDS-SD, que possui um alto nível de abstração, reduz o esforço de programação para desenvolver um novo fluxo de execução em relação à linguagem nativa da aplicação, como o nesC.

6.3 AVALIAÇÃO DE COMPORTAMENTO

O objetivo deste experimento é determinar se o código gerado utilizando a abordagem proposta apresenta anomalias ou se a implementação resultante tem o mesmo comportamento que os relatados anteriormente na literatura (Amis et al., 2000).

O simulador do TinyOS, o TOSSIM, utiliza 3 campos como parâmetros da topologia utilizada na simulação: (i) o ID do sensor origem, (ii) o ID do sensor destino e (iii) um valor de perda de potência do sinal em dBm (decibel miliwatt). Assim, foi necessário desenvolver um sistema para geração de topologias aleatórias utilizando esses 3 campos, como mostra a Figura 6.1. Para tornar a simulação mais realista, o sistema executa 3 etapas antes da geração dos topologias. O sistema esta disponível para download[2]. Esse sistema converte uma topologia de rede definida previamente para indicador de intensidade do sinal recebido.

A primeira etapa é a geração de coordenadas geográficas para cada dispositivo da rede. Para isso, ao executar o sistema, o desenvolvedor passa como parâmetros o tamanho da área que será monitorada (ex.: $200mx200m$) e o número de dispositivos. Então o sistema gera um arquivo com o ID de cada sensor e suas coordenadas (latitude e longitude) que serão utilizadas para construir a topologia de rede do TOSSIM.

Na segunda etapa, o sistema utiliza o arquivo com as coordenadas de cada sensor para calcular a distância entre eles e construir a topologia. Nessa etapa o desenvolvedor também deve definir qual a distância do raio de transmissão dos dispositivos (ex.: $20m$). A partir dai, utilizando a fórmula de *Haversine* (Chopde e Nichat, 2013) o sistema calcula a distância entre cada dispositivo e, caso a distância seja menor que o raio de transmissão, a próxima etapa é executada. Por fim, na terceira etapa, é gerado o valor da perda de potência do sinal utilizando como base a distância entre os dispositivos. Para cada par de identificadores é gerado um valor em dBM utilizando a fórmula dos modelos de propagação *Friis* e *Two Ray Ground* dos simuladores

[2] https://github.com/arordakowski/Gerador-de-RSSI

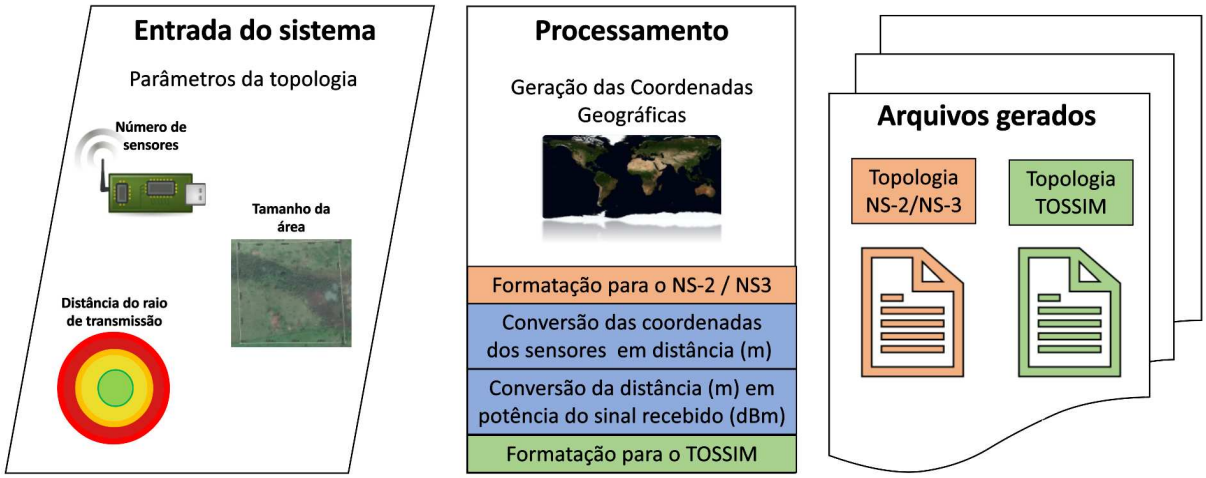


Figura 6.1: Sistema para geração de topologias do TOSSIM, NS-2 e NS-3.

NS-2 e NS-3[3]. O sistema de geração de RSSI é detalhado no Apêndice C. Os parâmetros de simulação são mostrados na Tabela 6.4.

Tabela 6.4: Parâmetros da Simulação

Parâmetros da Simulação	**Valor**
Areá Monitorada	$200mX200m$
Raio de Comunicação do Sensor	$20m$
d-saltos dos *Clusters*	2
Dispositivos na Rede	100, 200, 400 e 600

A simulação considerou uma região monitorada de $200X200$ metros quadrados, variando a densidade da rede. Foram consideradas redes com 100, 200, 400 e 600 dispositivos. O alcance do rádio de cada sensor no campo foi definido em 20 metros e o número máximo de saltos sem fio entre um nó e seu *Cluster-Head* foi definido em 2. Os resultados apresentados nesta seção correspondem à média de 35 simulações, com um intervalo de confiança de 95%.

6.3.1 Avaliação do número médio de *Cluster-Heads* de acordo com a densidade da rede

Neste experimento, o modelo MAX-MIN mostrou-se mais eficiente quando o número de nós da rede foi aumentado, uma vez que a quantidade de CHs não apresentou alterações significativas para as diferentes densidades. A Figura 6.2 mostra a comparação do número de CHs de acordo com a densidade da rede das aplicações MAX-MIN, LEACH e LCA.

O LEACH apresentou um resultado consistente com sua heurística, uma vez que o número de CHs esperados na rede é definido por um percentual. Neste experimento o percentual de *Cluster-Heads* na rede foi definido em 10%.

Por fim, o modelo de armazenamento LCA obteve um aumento maior com o crescimento da densidade da rede. Esses resultados são consistentes com os relatados no trabalho de Amis et al. (2000). Os detalhes de avaliação de cada modelo de armazenamento são descritos nas seções a seguir.

[3]Modelos de propagação: `https://www.nsnam.org/doxygen/group__propagation.html`

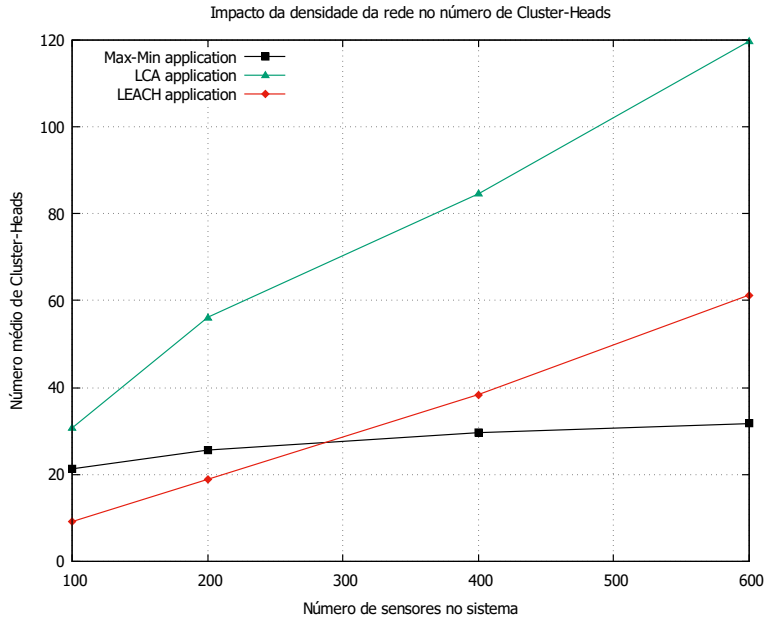


Figura 6.2: Impacto da densidade da rede no número de *Cluster – Heads*.

6.3.2 **Avaliação do modelo LCA**

Na Figura 6.3 são mostrados os resultados da simulação da aplicação LCA (Baker e Ephremides, 1981) gerada por SLEDS-SD.

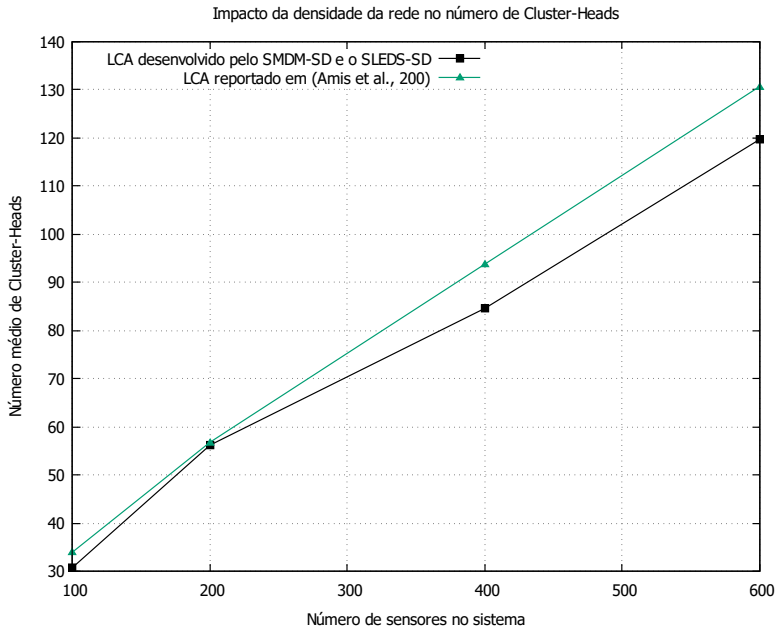


Figura 6.3: Impacto da densidade da rede no número de *Cluster-Heads* na aplicação LCA.

Nesta implementação do LCA, os sensores podem escolher os CHs com até dois saltos de distância. O mesmo parâmetro foi usado na implementação MAX-MIN. As simulações do LCA, quando comparadas com as simulações relatadas em Amis et al. (2000), mostram que o resultado da aplicação gerado por SLEDS-SD foram consistentes com os relatados. Embora os simuladores

utilizados nos experimentos sejam diferentes, é possível considerar que o comportamento da aplicação nos dois ambientes foi como o esperado.

6.3.3 **Avaliação do modelo LEACH**

O LEACH (Heinzelman et al., 2000), utiliza uma função probabilística para escolher uma porcentagem **p** de nós como *Cluster-Heads*. Na Figura 6.4 é mostrado o número médio de CHs com probabilidade de 10% em redes com 100, 200, 400 e 600 nós.

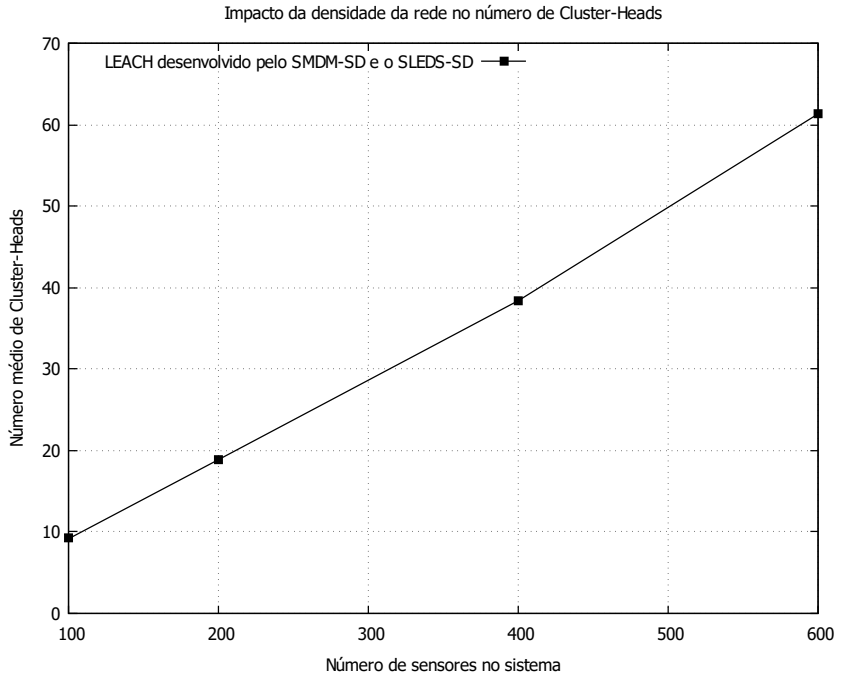


Figura 6.4: Impacto da densidade da rede no número de *Cluster-Heads* na aplicação LEACH.

Os resultados mostram que o código LEACH gerado pelo SLEDS-SD implementado no TinyOS está de acordo com o trabalho apresentado em (Heinzelman et al., 2000). O número de CHs segue a densidade da rede. Em simulações com densidades mais altas, o número de *Cluster-Heads* foi semelhante à porcentagem esperada de nós. Na rede com 400 nós, elegeu-se em média 38,4 nós, ou seja, 9,6%. Como na rede de 600 nós, a média de CHs foi de 61,3, sendo 10,21% dos dispositivos. Isso comprova que o código desenvolvido utilizando o SMDM-SD e a linguagem SLEDS-SD apresenta o comportamento esperado pela aplicação, visto que o fator determinante desse modelo de armazenamento está contido no código específico do componente de aplicação.

6.3.4 **Avaliação do modelo MAX-MIN**

Neste experimento, foi validada a exatidão da implementação MAX-MIN aplicando-a aos mesmos cenários de avaliação e parâmetros relatados em (Amis et al., 2000). Foram realizados três avaliações para avaliar se o comportamento da aplicação MAX-MIN implementada com apoio do modelo de desenvolvimento SMDM-SD e da linguagem SLEDS-SD são consistentes com os esperados. No primeiro experimento foi utilizada a mesma topologia de rede utilizada em (Amis et al., 2000) na simulação do MAX-MIN, com o objetivo de avaliar se a lógica da aplicação está correta, gerando os mesmos agrupamentos de sensores (*clusters*) ilustrados na Figura 6.5.

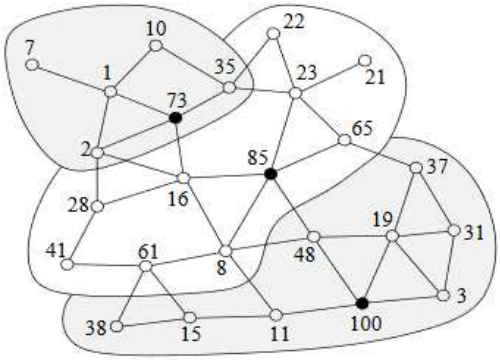


Figura 6.5: Cluster formado na execução de Max-Min (Amis et al., 2000).

Os resultados da aplicação desenvolvida pelo SMDM-SD são consistentes com o esperado, visto que os *clusters* gerados na simulação foram os mesmos que os apresentados na Figura 6.5.

Na Figura 6.6 é ilustrado o segundo experimento, o qual visa a avaliação do impacto da densidade da rede no número de *Cluster-Heads* na aplicação MAX-MIN. Nesse experimento foram geradas topologias aleatórias em cada densidade de rede, e em seguida calculada a média de 35 simulações. Os resultados mostram que o comportamento da aplicação gerada está próximo do esperado, conforme descrito em (Amis et al., 2000).

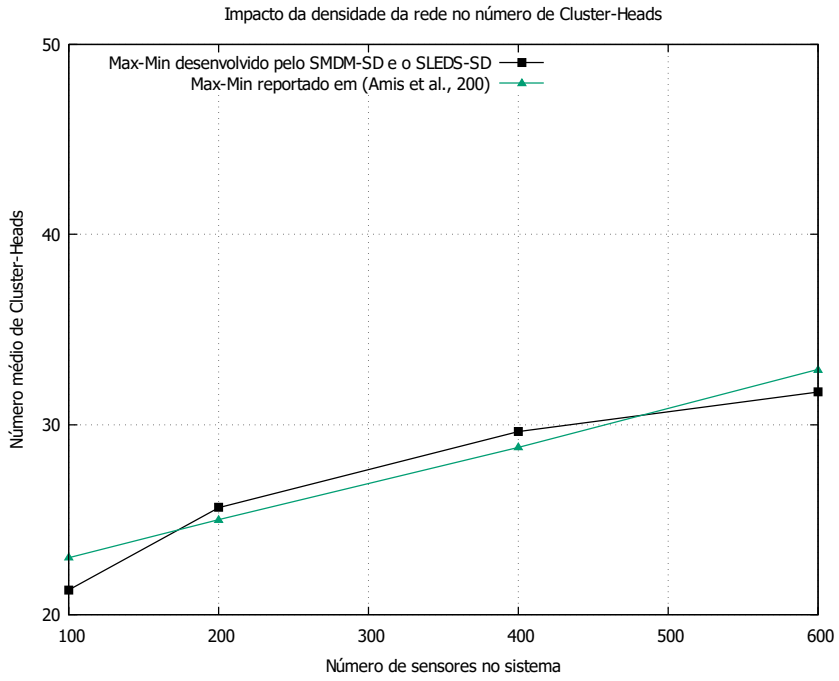


Figura 6.6: Impacto da densidade da rede no número de *Cluster-Heads* no aplicação Max-Min.

Pode ser observada uma diferença maior no cenário com poucos nós. Em cenários pouco povoados, a topologia da RSSF pode diferir significativamente, o que pode ter um impacto maior nos resultados. No entanto, conforme a densidade da rede aumenta, o número de CHs é bastante semelhante em ambos os ambientes. O número de CHs para a aplicação implementada por SLEDS-SD e o relatado no artigo que propôs MAX-MIN é próximo, validando assim o comportamento da implementação de MAX-MIN em SLEDS-SD.

No simulador do TinyOS, o Tossim, é possível controlar as perdas de mensagens definindo um limite, denominado valor de ganho, durante a execução do sistema. É útil fornecer perdas de mensagens durante as simulações para obter dados sobre o comportamento das aplicações em redes reais, visto que nestes ambientes alguns pacotes não são recebidos pelos destinatários. Com o objetivo de tornar o valor de ganho realístico com relação ao posicionamento dos sensores no ambiente, o sistema de geração de topologias gera o valor de ganho com base na distância entre as coordenadas do sensores transmissor e destinatário.

Na Figura 6.7 são ilustrados os resultados das simulações da aplicação MAX-MIN geradas pela plataforma SLEDS-SD. A consequência da perda de mensagens foi um aumento no número de CHs, pois algumas transmissões nas fases *Flood_Max* e *Flood_Min* não foram recebidas.

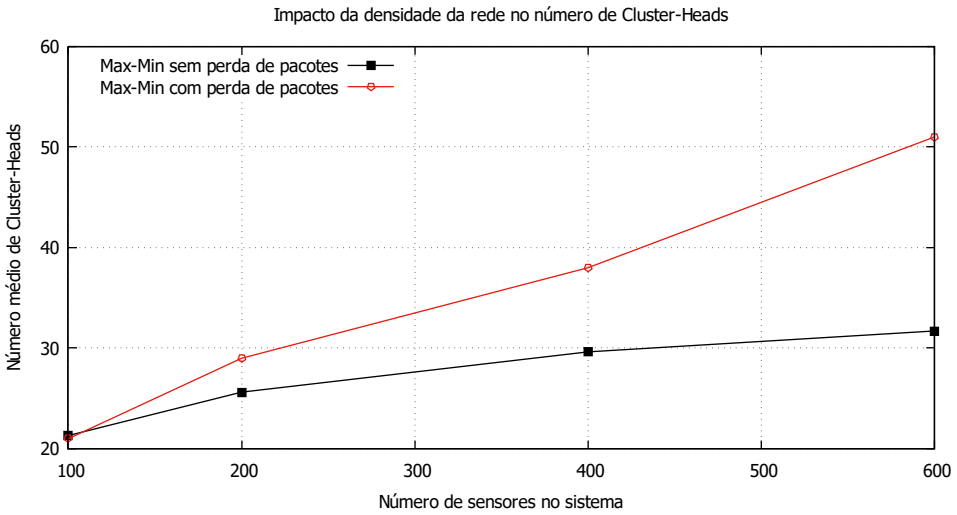


Figura 6.7: Impacto da densidade da rede com perda de pacotes no número de *Cluster-Heads* na aplicação Max-Min.

6.4 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados estudos experimentais com o objetivo de validar o ambiente de desenvolvimento proposto nesta dissertação. Em um primeiro momento foi analisado a capacidade de aproveitamento de código utilizando o SMDM-SD e a linguagem SLEDS-SD. Os estudos comprovaram que aplicações com fluxo de execução semelhante possuem cerca de 96% de reuso de código da linguagem SLEDS-SD. Porém, para aplicações com fluxo execução mais abrangentes, a taxa de aproveitamento foi próxima a 25%. Já na avaliação do reuso de código dos componentes do SMDM-SD, as aplicações semelhantes obtiveram aproximadamente 75% de aproveitamento de código, visto que boa parte dos componentes pré-desenvolvidos foram utilizados por ambas.

Na segunda etapa do estudo experimental foi avaliado o comportamento das aplicações, comparando os resultados com os apresentados na literatura. Os resultados mostram que o comportamento das aplicações geradas pelo SMDM-SD são consistentes com os esperados. Foi avaliado o número médio de *Cluster-Heads* em diversas densidade de rede. Mesmo utilizando simuladores diferentes, o número médio de CHs não se alterou de maneira significativa, validando assim o código desenvolvido pelo modelo proposto.

Os resultados apresentados nesse capítulo mostram que a estratégia para traduzir o código SLEDS-SD para nesC está correta e não impacta a qualidade do código gerado. Além

disso, o modelo utilizado para geração do código auxilia o desenvolvedor, pois a divisão em etapas lógicas com nomes explícitos facilita a compreensão do estado atual e da próxima etapa do fluxo de execução. No próximo capitulo são apresentadas as considerações finais e os trabalhos futuros.

7 CONCLUSÃO

Dado o crescente número de dispositivos nas RSSFs, bem com o volume de dados gerados, tornou-se fundamental o desenvolvimento de modelos de armazenamento que se apliquem aos fins específicos de cada rede. Entretanto, o desenvolvimento de tais modelos é uma tarefa complexa, tanto pela falta de flexibilidade no aproveitamento de código de outros modelos, quanto pela dificuldade de expressar seu fluxo de execução.

Nessa dissertação foi investigada a aplicação de um método formal que auxilie no aproveitamento de código e na especificação do fluxo de execução de modelos de armazenamento para RSSFs. Assim, foi elaborado o **modelo de desenvolvimento SMDM-SD** com o objetivo de apoiar a implementação de novos sistemas de armazenamento, baseados em dois princípios: (i) especificação do fluxo de execução como máquina de estados e (ii) implementação de componentes que prestam serviços reutilizáveis. Para uma implementação rápida baseada em máquina de estados, foi introduzida uma **linguagem específica de domínio**, chamada **SLEDS-SD**, com primitivas que se assemelham a máquina de estados com tipos de transições usados na especificação. Um tradutor de SLEDS-SD para nesC foi desenvolvido para gerar código a ser instalado em RSSFs reais.

Um estudo experimental que analisa a implementação dos modelos de armazenamento LCA, LEACH e MAX-MIN seguindo a abordagem proposta mostrou que os programas SLEDS-SD são cerca de 40 % menores do que o código nesC gerado, o que indica que nossa abordagem reduz o esforço de desenvolvimento. Além disso, a avaliação da reutilização de código mostrou que aplicativos com um fluxo de execução semelhantes possuem mais de 95% de linhas idênticas de código SLEDS-SD. A taxa de reutilização de código de componentes implementados no nesC variou de 32% a 78%. Também foi realizado um estudo sobre o comportamento das aplicações desenvolvidas utilizando a abordagem apresentada nesta dissertação, em relação ao relatado em outros estudos na literatura. Para a simulação, foi implementado um sistema de geração de topologias aleatórias juntamente com a indicação de força do sinal recebido (RSSI). Nessa avaliação foi considerado o número médio de CHs de acordo com a densidade da rede. Os resultados da execução da aplicação foram próximos aos relatados em outros artigos que utilizaram diferentes simuladores. Isso mostra a exatidão e confiabilidade da proposta.

7.1 LISTA DE PUBLICAÇÕES RELACIONADAS À DISSERTAÇÃO

- O SMDM-SD para geração de código em nesC como um artigo curto do **Simpósio Brasileiro de Banco de Dados (SBBD)** (Ordakowski et al., 2019a);
- A proposta de pesquisa no **Workshop de Teses e Dissertação de Banco de Dados (WTDBD) do SBBD** (Ordakowski et al., 2019b);
- Uma amostra resumida de ambos os trabalhos também foi apresentada na primeira edição do **Workshop em Estudos de Computação** (WEC)[1] do Programa de Pós-graduação em Informática da UFPR;
- O SMDM-SD, o SLEDS-SD, bem como as análises apresentadas na dissertação, foram aceitos para a publicação como artigo curto do ***International Conference on Cloud Computing and Services Science*** **(CLOSER)** (Hara et al., 2021).

[1]Link do workshop: `http://www.inf.ufpr.br/wec/`

7.2 TRABALHOS FUTUROS

Esta dissertação apresenta contribuições para profissionais que atuam no desenvolvimento de sistemas de armazenamento em RSSF e para estudiosos da área. Como trabalho futuro, pretende-se estender o SMDM-SD para gerar código tanto para simuladores de rede, como o NS-3, quanto para outras plataformas de RSSF e IoT, como Contiki e RIOT. Com essas extensões, será possível usar a mesma especificação de alto nível para primeiro avaliar um modelo de armazenamento usando simuladores e, em seguida, implantar o aplicativo em diferentes plataformas reais com menos esforço.

Outro trabalho futuro é investigar a viabilidade de adaptar o modelo de desenvolvimento SMDM-SD e a linguagem SLEDS-SD para apoiar a especificação e implementação do **coordenador principal** dos dispositivos em RSSF. Esse coordenador é o responsável por orquestrar as aplicações instaladas em cada sensor, indicando quando cada aplicação deve ser executada, bem como fornecendo os parâmetros exigidos por elas. Por fim, também é planejado verificar se é possível adaptar o SLEDS-SD para o desenvolvimento de aplicações em outros domínios(Boubiche et al., 2018).

REFERÊNCIAS

Abbasi, A. A. e Younis, M. (2007). A survey on clustering algorithms for wireless sensor networks. *Computer Communications*, 30(14-15):2826–2841.

Ahmed, A. I. A., Gani, A., Ab Hamid, S. H., Abdelmaboud, A., Syed, H. J., Mohamed, R. A. A. H. e Ali, I. (2019). Service management for iot: requirements, taxonomy, recent advances and open research challenges. *IEEE Access*, 7:155472–155488.

Aho, A., Lam, M., Sethi, R. e Ullman, J. (2007). Compiladores: princípios, técnicas e ferramentas, 2ª edição.

Aho, A. V., Sethi, R. e Ullman, J. D. (1998). *Compilers: Principles, Techniques and Tools*. Pearson Education.

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y. e Cayirci, E. (2002). Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422.

Amaxilatis, D., Chatzigiannakis, I., Koninis, C. e Pyrgelis, A. (2011). Component based clustering in wireless sensor networks. *arXiv preprint arXiv:1105.3864*.

Amis, A. D., Prakash, R., Vuong, T. H. e Huynh, D. T. (2000). Max-min d-cluster formation in wireless ad hoc networks. Em *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, volume 1, páginas 32–41. IEEE.

Baker, D. J. e Ephremides, A. (1981). A distributed algorithm for organizing mobile radio telecommunication networks. Em *ICDCS*, páginas 476–483.

Basu, A., Mounier, L., Poulhies, M., Pulou, J. e Sifakis, J. (2007). Using bip for modeling and verification of networked systems–a case study on tinyos-based networks. Em *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007)*, páginas 257–260. IEEE.

Baumgartner, T., Chatzigiannakis, I., Fekete, S., Koninis, C., Kroller, A. e Pyrgelis, A. (2010). Wiselib: A generic algorithm library for heterogeneous sensor networks. Em *European Conference on Wireless Sensor Networks*, páginas 162–177. Springer.

Boubiche, S., Boubiche, D. E., Bilami, A. e Toral-Cruz, H. (2018). Big data challenges and data aggregation strategies in wireless sensor networks. *IEEE Access*, 6:20558–20571.

Braga, M. d. L. (2012). Geração automática de código para redes de sensores sem fio usando communicating x-machine. Dissertação de Mestrado, Programa de Pós-graduação em Engenharia Elétrica - Universidade Federal do Amazonas, Manaus.

Carrero, M. A. (2018). *Geração de Códigos de Simulação para Armazenamento de Dados em Redes de Sensores sem Fio*. Tese de doutorado, Universidade Federal do Paraná, Brasil.

Carrero, M. A., Musicante, M. A., dos Santos, A. L. e Hara, C. S. (2017). A reusable component-based model for wsn storage simulation. Em *Proceedings of the 13th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, páginas 31–38. ACM.

Carrero, M. A., Musicante, M. A., dos Santos, A. L. e Hara, C. S. (2018). Sleds: A dsl for data-centric storage on wireless sensor networks. Em *Workshop on Big Social Data and Urban Computing*, páginas 74–89. Springer.

Carrero, M. A., Musicante, M. A., dos Santos, A. L. e Hara, C. S. (2019). A DSL for WSN software components coordination. *Information Systems*, página 101461.

Cattani, M., Zuniga, M., Loukas, A. e Langendoen, K. (2014). Lightweight neighborhood cardinality estimation in dynamic wireless networks. Em *Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, IPSN '14, página 179–189. IEEE Press.

Chaudron, M., Larsson, S. e Crnkovic, I. (2005). Component-based development process and component lifecycle. *Journal of Computing and Information Technology*, 13(4):321–327.

Chen, K. (2001). *Programming Open Service Gateways with Java Embedded Server Technology*. Addison-Wesley Longman Publishing Co., Inc.

Chopde, N. R. e Nichat, M. K. (2013). Landmark based shortest path detection by using a* and haversine formula. *International Journal of Innovative Research in Computer and Communication Engineering*, 1(2):298–302.

Costa, R. A. G. d. (2011). *Utilização de data warehouses para gerenciar dados de redes de sensores sem fio que monitoram polinizadores*. Tese de doutorado, Universidade de São Paulo.

Dastjerdi, A. V. e Buyya, R. (2016). Fog computing: Helping the internet of things realize its potential. *Computer*, 49(8):112–116.

Dunkels, A., Gronvall, B. e Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. Em *29th Annual IEEE International Conference on Local Computer Networks*, páginas 455–462.

Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E. e Culler, D. (2003). The nesc language: A holistic approach to networked embedded systems. *Acm Sigplan Notices*, 38(5):1–11.

Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E. e Culler, D. (2014). The nesc language: A holistic approach to networked embedded systems. *Acm Sigplan Notices*, 49(4):41–51.

Hara, C. S., Ordakowski, A. R. e Carrero, M. A. (2021). Development of wireless sensor networks applications with state-based orchestration. Em *International Conference on Cloud Computing and Services Science*. Springer.

Heinis, T. B. (2019). *Enabling the Development of Value-Adding Internet of Things Applications in the Manufacturing Industries*. Tese de doutorado, ETH Zurich.

Heinzelman, W. R., Chandrakasan, A. e Balakrishnan, H. (2000). Energy-efficient communication protocol for wireless microsensor networks. Em *Proceedings of the 33rd annual Hawaii international conference on system sciences*, páginas 10–pp. IEEE.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. e Pister, K. (2000). System architecture directions for networked sensors. Em *ACM SIGOPS operating systems review*, volume 34, páginas 93–104. ACM.

Hussein, M., Li, S. e Radermacher, A. (2017). Model-driven development of adaptive iot systems. Em *MODELS (Satellite Events)*, páginas 17–23.

Khan, I., Belqasmi, F., Glitho, R., Crespi, N., Morrow, M. e Polakos, P. (2015). Wireless sensor network virtualization: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):553–576.

Lau, K.-K. e di Cola, S. (2017). *An Introduction to Component-Based Software Development*. World Scientific.

Lekidis, A., Stachtiari, E., Katsaros, P., Bozga, M. e Georgiadis, C. K. (2018). Model-based design of iot systems with the bip component framework. *Software: Practice and Experience*, 48(6):1167–1194.

Lin, J., Yu, W., Zhang, N., Yang, X., Zhang, H. e Zhao, W. (2017). A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things Journal*, 4(5):1125–1142.

Louden, K. C. e Silva, F. S. C. (2004). *Compiladores-Princípios e Práticas*. Cengage Learning Editores.

Loureiro, A. A., Nogueira, J. M. S., Ruiz, L. B., Mini, R. A. d. F., Nakamura, E. F. e Figueiredo, C. M. S. (2003). Redes de sensores sem fio. Em *Simpósio Brasileiro de Redes de Computadores (SBRC)*, páginas 179–226. sn.

Lucrédio, D. (2009). *Uma abordagem orientada a modelos para reutilização de software*. Tese de doutorado, Instituto de Ciências Matemáticas e de Computação Universidade de São Paulo.

Myklebust, G. (1996). The avr microcontroller and c compiler co-design. *ATMEL Development Center, Trondheim, Norway*.

Ordakowski, A., Carrero, M. A., Musicante, M., dos Santos, A. e Hara, C. (2019a). Desenvolvimento de modelos de armazenamento em sensores com reutilização de código. Em *Simpósio Brasileiro de Banco de Dados 2019 - Short and Vision and Industrial Papers*, Fortaleza, Ceará.

Ordakowski, A., Hara, C. e Carrero, M. A. (2019b). Uma dsl para coordenação de componentes de armazenamento em redes de sensores sem fio. Em *Simpósio Brasileiro de Banco de Dados 2019 - WTDBD*, Fortaleza, Ceará.

Ouadjaout, A., Lasla, N., Bagaa, M. e Badache, N. (2014). Static analysis of device drivers in tinyos. Em *IPSN-14 Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, páginas 297–298. IEEE.

Rashid, B. e Rehmani, M. H. (2016). Applications of wireless sensor networks for urban areas: A survey. *Journal of Network and Computer Applications*, 60:192 – 219.

Romadi, R. e Berbia, H. (2008). Wireless sensor network: A specification method based on reactive decisional agents. Em *3rd International Conference on Information and Communication Technologies: From Theory to Applications*, páginas 1–5.

Ruiz, L. B. (2003). *MANÁ: uma arquitetura para gerenciamento de redes de sensores sem Fio.* Tese de doutorado, Universidade Federal de Minas Gerais, Brasil.

Sadilek, D. A. (2007). *Prototyping and simulating domain-specific languages for wireless sensor networks*. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät .

Salman, A. J. e Al-Yasiri, A. (2016). Sennet: a programming toolkit to develop wireless sensor network applications. Em *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, páginas 1–7. IEEE.

Satyanarayanan, M. (2017). The emergence of edge computing. *Computer*, 50(1):30–39.

Schiller, J., Liers, A., Ritter, H., Winter, R. e Voigt, T. (2005). Scatterweb-low power sensor nodes and energy aware routing. Em *Proceedings of the 38th annual Hawaii international conference on system sciences*. IEEE.

Shelke, R., Kulkarni, G., Sutar, R., Bhore, P., Nilesh, D. e Belsare, S. (2013). Energy management in wireless sensor network. Em *2013 UKSim 15th International Conference on Computer Modelling and Simulation*, páginas 668–671. IEEE.

Singh, J., Mishra, A. K. et al. (2015). Clustering algorithms for wireless sensor networks: a review. Em *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, páginas 637–642. IEEE.

Soukaras, D., Patel, P., Song, H. e Chaudhary, S. (2015). Iotsuite: a toolsuite for prototyping internet of things applications. Em *The 4th International Workshop on Computing and Networking for Internet of Things (ComNet-IoT), co-located with 16th International Conference on Distributed Computing and Networking (ICDCN)*, página 6.

Taherkordi, A., Johansen, C., Eliassen, F. e Römer, K. (2015). Tokenit: Designing state-driven embedded systems through tokenized transitions. Em *2015 International Conference on Distributed Computing in Sensor Systems*, páginas 52–61. IEEE.

Taherkordi, A., Loiret, F., Rouvoy, R. e Eliassen, F. (2011). A generic component-based approach for programming, composing and tuning sensor software. *The Computer Journal*, 54(8):1248–1266.

Vara, J. M. e Marcos, E. (2012). A framework for model-driven development of information systems: Technical decisions and lessons learned. *Journal of Systems and Software*, 85(10):2368–2384.

APÊNDICE A – SINTAXE DA LINGUAGEM SLEDS SD

A sintaxe da linguagem SLEDS-SD é apresentada a seguir na Tabela A.1. A linguagem utiliza construções semelhantes a linguagens de programação conhecidas, como o C, porém também inclui termos relativos ao método formal utilizado na especificação dos sistemas, a máquina de estados.

Use	::=	**USE** Identifier **AS** Identifier ;
MessageType	::=	**MESSAGE TYPE** { Identifier (, Identifier)* } ;
Message	::=	**MESSAGE** Identifier { VarList } ;
Program	::=	**PROGRAM** Identifier ((VarList)*) { (**CONST** Identifier = (NumLiteral | StrLiteral);)* (VarList)* StateDef* }
VarList	::=	((SimpleType | ListType) Identifier (, Identifier)* ;)*
SimpleType	::=	**INT** | **REAL** | **STRING** | **BOOL**
ListType	::=	**LIST** <SimpleType >Identifier [IntLiteral] ;
StateDef	::=	**STATE** Identifier (VarList) { ActionList }
State	::=	Identifier (ExpList?) | **EXIT**
ActionList	::=	Action (Action)*
Action	::=	**NEXTSTATE** State; | **SEND** (Identifier, Identifiern Identifier, Identifier) | **BROADCAST** (Identifier, Identifier, Identifier) | **ON RECV** (Identifier, Identifier, Identifier) {ActionList } | **ON RECVBROADCAST** (Identifier, Identifier, Identifier) {ActionList } | **DURING** (Exp) **ON RECV** (Identifier, Identifier, Identifier, Identifier) { ActionList } | **DURING** (Exp) **ON RECVBROADCAST** (Identifier, Identifier Identifier) { ActionList } | **WHILE** (Exp) { ActionList } | **FOR** (Assigment; Exp; Exp) { ActionList } | **IF** (Exp) { ActionList } (**ELSE** { ActionList })? | Assigment ; | MethodCall ;
MethodCall	::=	Exp
Assigment	::=	Identifier = Exp
Exp	::=	Exp->Exp | Exp . Exp | Exp(Exp?) | ExprMat | ExprBool | ListFunc | Identifier | NumLiteral | StrLiteral
ListFunc	::=	Identifier.**INSERT**(Exp) | Identifier.**SIZE**( ) | Identifier.**NEXT**( ) | Identifier.**FIRST**( ) | Identifier.**LAST**( ) | Identifier.**REMOVE**( )

Tabela A.1: Sintaxe do SLEDS-SD

Para realizar a importação de componentes para a aplicação é utilizado o `USE`. Também é possível utilizar os componentes com um identificador diferente, utilizando o `AS` acompanhado do novo identificador. Podem ser importados componentes existentes, como os de biblioteca,

e componentes que serão desenvolvidos, como os componentes de aplicação em alguns casos. Quando o componente não existe, na etapa de geração de código, a interface dos componentes é gerada, juntamente com sua funções.

Para identificar os tipos de mensagens, no SLEDS-SD é utilizado o `MESSAGE TYPE`. Esses identificadores funcionam como um enumerado na linguagem C, e são um parâmetro das *Actions* que envolvem troca de mensagens. Para construir a estrutura das mensagens, é declarado o `MESSAGE` acompanhado de um identificador e os parâmetros da mensagem. Posteriormente a estrutura será utilizada pra declarar um pacote, que é a carga útil da mensagem, sendo um dos parâmetros de *Actions* relacionadas a transmissão e recepção de mensagens.

A parte lógica da aplicação é iniciada após a declaração de um novo programa, por meio do `PROGRAM`. A declaração de um programa possui um identificador e uma lista de variáveis como parâmetro. A seguir, são declaradas as constantes da aplicação por meio do `CONST`, que armazena um valor literal para um identificador. Após isso são declaradas as variáveis, que podem ser dos tipos inteiro `INT`, real `REAL`, string `STRING`, booleano `BOLL` e lista `LIST`.

A execução da aplicação acontece dentro de blocos de código lógico, denominados **estados**. Para declarar um estado utiliza-se o `STATE`, acompanhado do identificador do estado, uma lista de variáveis como parâmetro, e uma lista de ações. As possíveis ações da lista são:

- `nextState`: ação que faz uma transição para outro estado.
- `send`: ação para transmitir uma mensagem a um sensor específico.
- `broadcast`: ação para transmitir uma mensagem de broadcast.
- `on recv`: ação para receber mensagens específica.
- `on recvBroadcast`: ação para receber mensagens broadcast.
- `during`: ação para acionar um cronômetro.
- `during on recv`: ação para receber mensagens específicas durante um período de tempo.
- `during on recvBroadcast`: ação para receber mensagens broadcast durante um período de tempo.
- `Assignment`: ação para atribuir um valor a um identificador.
- `Method-call`: ação que executa uma chamada de método.

As expressões da gramática envolvem as chamadas de métodos, a parte lógica de dos laços de repetição e condicionais, e funções ligadas a lista. O Apêndice B apresenta um exemplo de programa em SLEDS-SD.

APÊNDICE B – CÓDIGO DA APLICAÇÃO MAX-MIN EM SLEDS-SD

A aplicação Max-Min pode ser dividida em 3 fases, a inundação (`FloodMax` e `FloodMin`), a formação dos *clusters* (`SelectCH`, `ClusterFormation` e `StoreMembers`), e a formação dos *gateways*. Na Listagem B.1 é apresentado o código em SLEDS-SD das fases de inundação e formação do cluster.

```
1  \\Declarando as importações
2  use compSensor as ComponentesSensor;
3  use compCluster as ComponentesCluster;
4  use compLibMSG as ComponentesLibMessage;
5  use compLibAggregation as ComponentesLibAggregation;
6  use compList;
7
8  \\Declaração de tipos de mensagem
9  message type {MAXWINNER, MINWINNER, CMANNOUNCE};
10
11 \\Declarações de estruturas de mensagem
12 message msgMaxWinner { int winnerID; };
13 message msgMinWinner { int winnerID; };
14 message msgCmAnnounce { int myID; };
15
16 Program Coordinator () {                    //Declaração de um programa SLEDS−SD
17
18 msgMaxWinner pktMaxWinner;                  //Declaração dos pacotes de mensagem
19 msgMinWinner pktMinWinner;
20 msgCmAnnounce pktCmAnnounce;
21
22 const dHops=2;                              //Declarações de variáveis e constantes
23 const tFloodMAX=25;
24 const tFloodMIN=25;
25 const tCluster =25;
26 int round, msgId, myID, winnerID, myCH;
27  list <int> ListMaxWinner[20];
28  list <int> ListMinWinner[20];
29
30 STATE INI(){                                //Declaração do estaodo INI
31     myID=compSensor−>getSensorId();          //Assigment
32     winnerID=myID;
33      nextState FloodMax(winnerID); }         //Transição de estados
34
35 STATE FloodMax(int maxWinnerID) {
36     msgId = compLibMSG −>GetNextMsgId();     //Assigment e uso do componente de MSG
37     pktMaxWinner−>winnerID = maxWinnerID;    // Inserir valor no pacote
38      broadcast (MAXWINNER , msgId, pktMaxWinner); //Enviar mensagem Broadcast
39      nextState StoreMaxWinner(); }
40
41 STATE StoreMaxWinner () {
42     during (tFloodMAX) on recv(MAXWINNER, msgID, pktMaxWinner) { //Action During on recv
```

```
43          ListMaxWinner.insert(pktMaxWinner->winnerID); }          // Inserir valor na lista
44      nextState    SelectMaxWinner (); }
45
46  STATE SelectMaxWinner () {
47      winnerID = compLibAggregation->MAX(ListMaxWinner);
48      if (round < dHops) {                                          //Action if
49          round++;
50          nextState    FloodMax( winnerID );
51      } else {                                                      //Action else
52          round=0;
53          nextState   FloodMin( winnerID ); } }
54
55  STATE FloodMin ( int minWinnerID ) {
56        msgId = compLibMSG->GetNextMsgId();
57        pktMinWinner->winnerID = minWinnerID;
58        broadcast(MINWINNER , msgId, pktMinWinner);
59        nextState    StoreMinWinner (); }
60
61  STATE StoreMinWinner ( ) {
62        during (tFloodMIN) on recv(MINWINNER, msgID, pktMinWinner) {
63            ListMinWinner( pktMinWinner->winnerID ); }
64        nextState    SelectMinWinner (); }
65
66  STATE SelectMinWinner ( ) {
67        winnerID = compLibAggregation->MIN( ListMinWinner );
68        if (round < dHops) {
69            round++;
70            nextState    FloodMin( winnerID );
71        } else {
72            round=0;
73            nextState   SelectCH( ); } }
74
75  STATE SelectCH ( ) {
76          myCH = compCluster->selectCH(ListMaxWinner, ListMinWinner); //Componente Cluster
77          nextState    ClusterFormation ( );   }
78
79  STATE ClusterFormation ( ) {
80      pktCmAnnounce->myID = myID;
81      pktCmAnnounce->myCH = myCH;
82      msgId = compLibMSG->GetNextMsgId();
83      compLibMSG->addSeenMsg(CMANNOUNCE, msgId);
84        broadcast(CMANNOUNCE, msgId, pktCmAnnounce);
85        nextState    StoreMembers ( ); }
86
87  STATE StoreMembers ( ) {
88      compSensor->role(1);                  // 1: Cluster-Member
89      during ( tCluster ) on recv(CMANNOUNCE, msgID, pktCmAnnounce) {
90          if ( pktCmAnnounce->myCH == myID){
91              compSensor->role(2);       // 2: Cluster-Head
92              compCluster->setMembers( pktCmAnnounce->myID );
93          } else if ( pktCmAnnounce->myCH == myCH
```

```
94                  && compLibMSG->seenMsg(CMANNOUNCE, msgID)==FALSE) {
95                    compLibMSG->addSeenMsg(CMANNOUNCE, msgID);
96                      broadcast(CMANNOUNCE, msgID, pktCmAnnounce);}}
97      nextState    exit ;  }
98  }
```

Listagem B.1: Implementação das etapas de formação do *cluster* da aplicação Max-Min em SLEDS-SD

No inicio são realizadas as importações de componentes (linhas 2-6), declaração de tipos de mensagem (linha 9), e declarações de estruturas de mensagens (linhas 12-14). A parte lógica da aplicação ocorre após a declarar um novo programa (linha 16). Nessa parte são atribuídas as estruturas das mensagens aos pacotes (linhas 18-20), e são declaradas as constantes e as variáveis (linhas 22-28).

O estado `INI` realiza a ação de atribuição de valor da variável *myID*, que recebe o valor retornado da função *getSensorId( )* do componente *CompSensor*. Em seguida, ocorre uma nova atribuição de valor, em que a variável *winnerID* recebe o valor de *myID*. Por fim, é realizado uma transição de estados para `FloodMax` com a variável *winnerID* como parâmetro (linha 33).

O estado `FloodMax` recebe um inteiro *maxWinnerID* como parâmetro (linha 35). Nesse estado é realizado uma atribuição de valor na variável *msgId*. A variável *winnerID* do pacote *pktMaxWinner* recebe o valor de *maxWinnerID* recebido como parâmetro. Em seguida o pacote é transmitido por meio de um broadcast (linha 38), e ocorre a transição de estados para o `StoreMaxWinner`.

No estado `StoreMaxWinner` as mensagens do tipo *MAXWINNER* são recebidas durante o tempo *tFloodMAX* (linha 42). O conteúdo das mensagens recebidas é inserido na lista *ListNaxWinner*. No final do temporizador é realizado uma transição de estados para o `SelectMaxWinner`. Nesse estado (`SelectMaxWinner`), é atribuído para a variável *winnerID* o maior valor da lista *ListMaxWinner*, retornado através da função *Max* do componente *compLibAggregation*. Em seguida, é verificado se o número de *rounds* é menor que o definido pela constante *dHops* (linha 48). Se for menor, o valor de *rounds* é incrementado e o estado `FloodMax` é executado novamente (linhas 49 e 50). Se não for, o valor de *round* passa a ser 0, e ocorre a transição para o estado `FloodMin` (linhas 52 e 53). Os estados `FloodMin` , `StoreMinWinner` e `SelectMinWinner` funcionam da mesma maneira apresentada anteriormente, com a transmissão e seleção do menor valor na lista*ListMinWinner* (linhas 55-73).

No estado `SelectCH` (linha 75), a variável *myCH* recebe o valor retornado da função *selectCH* do componente *compCluster* (linha 76). Essa função utiliza os critérios específicos do algoritmo Max-Min para eleger um *Cluster-Head*. em seguida é realizado a transição para o estado `ClusterFormation`. No estado `ClusterFormation` os pacotes *pktCmAnnounce* são enviados através de um *broadcast* para todos os sensores no raio de transmissão, e em seguida é realizado a transição para o estado `StoreMembers`.

No `StoreMembers` (linha 88), o sensor se declara um *cluster-member* (linha88). Durante o tempo *tCluster* são recebidas mensagens do tipo CMANNOUNCE (linha 89). Se a variável *myCH* do pacote *pktCMAnnounce* recebido for igual ao *myID*, o sensor atualiza sua função para *Cluster-Head*, e envia o valor da variável *myID* do pacote *pktCmAnnounce* para a função *setMembers* do componente *compCluster* (linhas 90-92). Se for diferente, o sensor verifica se o valor de *pktCmAnnounce->myCH* é igual ao *myCH* e se o valor retornado da função *compLibMSG->seenMsg* é falso. Caso a resposta seja positiva, o sensor envia o identificador do tipo de mensagem e o identificador da mensagem para a função *compLibMSG->addSeenMSG*. Em seguida, é realizado um *broadcast* do pacote *pktCmAnnounce* recebido. Ao fim do temporizador *tCluster*, a aplicação realiza uma transição para o estado `exit`.

APÊNDICE C – SISTEMA DE GERAÇÃO DO INDICADOR DE POTÊNCIA DO SINAL RECEBIDO

O simulador do TinyOS na sua versão mais atual, o TOSSIM, não possui uma funcionalidade relacionada ao indicador de potência do sinal recebido (RSSI). Portanto, para executar aplicações que utilizam essa funcionalidade, como o LCA e o LEACH, foi desenvolvido um sistema de geração de RSSI para o simulador TOSSIM.

O sistema esta disponível para download no seguinte link `https://github.com/arordakowski/Gerador-de-RSSI`. Esse sistema converte uma topologia de rede definida previamente para indicador de intensidade do sinal recebido. A primeira etapa é preparar o arquivo de entrada do sistema, denominado `coord.txt`. Esse arquivo possui os identificadores dos sensores, e suas coordenadas geográficas. Essas coordenadas serão utilizadas como parâmetros para a conversão da distância entre dois sensores em metros, para posteriormente realizar a busca pela perda de potência do sinal do rádio. Para automatizar essa tarefa, o sistema gera coordenadas aleatórias recebendo como parâmetros o tamanho da areá em metros e o número de sensores na rede. A topologia gerada nesta primeira etapa do processo é constituída pelo identificador do mote (sensor), juntamente com sua latitude e longitude.

ID	LATITUDE	LONGITUDE
1	-10.8054069	-55.4592031
2	-10.805567	-55.4591173
3	-10.8056282	-55.4590435
4	-10.8055782	-55.4590797
5	-10.8053279	-55.4584843
6	-10.8056151	-55.459077
7	-10.8054359	-55.4585164

Tabela C.1: Tabela com entradas do sistemas de geração de RSSI.

Em seguida, o sistema calcula a distância em metros entre as coordenadas dos IDs utilizando a fórmula de Haversine [1]. A conversão da distância em RSSI utiliza os mesmos modelos de perda propagação dos simuladores NS-2 e NS-3, o Modelo de espaço livre e o Modelo de reflexão do solo em dois raios.

Modelo de espaço livre: O modelo de propagação de espaço livre assume a condição ideal de propagação de que haja apenas um caminho claro da linha de visão entre o transmissor e o receptor.

$$Prd \ \frac{PtGtGrX^2}{4\Pi^2 d^2 L}$$

onde Pt é a potência do sinal transmitido, Gt e Gr são os ganhos de antena do transmissor e do receptor, respectivamente. L é o comprimento de onda de transmissão. O modelo de espaço livre representa basicamente o alcance da comunicação como um círculo ao redor do transmissor. Se um receptor estiver dentro do círculo, ele receberá todos os pacotes. Caso contrário, ele perde todos os pacotes. Esse modelo só é utilizado quando os sensores estão próximos, visto que o modelo de reflexão do solo em dois raios traz valores mais condizentes com a realidade.

[1] `https://www.jstor.org/stable/2309088?seq=1`

Modelo de reflexão do solo em dois raios: Um único caminho de linha de visão entre dois nós raramente é o único meio de propagação. O modelo de reflexão do solo em dois raios considera o caminho direto e o caminho de reflexão do solo.

$$Prd \ \frac{PtGtGrHt^2Hr^2}{d^4L}$$

onde Pt é a potência do sinal transmitido, Gt e Gr são os ganhos de antena do transmissor e do receptor, respectivamente. L é o comprimento de onda de transmissão e ht e hr são as alturas das antenas transmissoras e receptoras, respectivamente. A equação mostra uma diminuição rápida da potência do RSSI de acordo com o aumento da distância entre os sensores. No entanto, o modelo de dois raios não fornece um bom resultado para uma curta distância devido à oscilação causada pela combinação construtiva e destrutiva dos dois raios. Em vez disso, o modelo de espaço livre ainda é usado.

Os valores retornados pelos modelo de propagação utilizados retorna o RSSI em miliwatt (mW). A leitura no TOSSIM é em dBm (decibel miliwatt), portanto o sistema realiza a conversão de mW em dBm. A Tabela C.2 apresenta os resultados de cada etapa do processamento.

ORIGEM	**DESTINO**	**DISTÂNCIA**	**RSSI (mW)**	**RSSI (dBm)**
1	2	20.118266m	0.000000001443676	-88.41dBm
1	3	30.156297m	0.000000000642532	-91.92dBm
1	4	23.333997m	0.000000001073180	-89.69dBm
1	6	26.938005m	0.000000000805230	-90.94dBm
2	3	10.549141m	0.000000005250692	-82.80dBm
2	4	4.291476m	0.000000031727617	-74.99dBm
2	6	6.926843m	0.000000012178114	-79.14dB
3	4	6.822315m	0.000000012554148	-79.01dBm
3	6	3.938271m	0.000000037673804	-74.24dBm
4	6	4.113677m	0.000000034529507	-74.62dBm
5	7	12.510390m	0.000000003733439	-84.28dBm

Tabela C.2: Tabela com a saída do sistema de geração de RSSI.

Por fim, é gerado o arquivo de saída com a formatação de cada simulador. No NS-2 / NS-3 a saída é um arquivo com a inicialização dos nós e inserção das coordenadas X, Y e Z de cada sensor. O arquivo de saída do TOSSIM é formatado como um grafo com peso, onde os vértices são os sensores remetente e destinatário, e o peso das arestas é a perda de sinal em dBm.